Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Blaž Berglez

Primerjava ogrodij za hkraten navzkrižni razvoj mobilnih aplikacij za več platform

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Aleš Smrdel

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V svetu postajajo vse bolj pomembne aplikacije za mobilne naprave. Na mobilnih napravah v zadnjem času prevladujeta dve platformi. Razvoj mobilne aplikacije za čim širši krog uporabnikov tako lahko zahteva razvoj aplikacij za vsaj dve platformi, lahko pa tudi več. V okviru diplomskega dela raziščite mogoče pristope k razvoju mobilne aplikacije za več platform in jih predstavite. Nato si zamislite aplikacijo, ki bo vključevala funkcionalnosti tipične mobilne aplikacije in z nekaj izbranimi ogrodji za hkraten navzkrižni razvoj mobilnih aplikacij za več platform implementirajte to aplikacijo. Predstavite postopek izdelava aplikacije s posameznimi ogrodji. Uporabljena ogrodja primerjajte, primerjajte pa tudi aplikacije na različnih platformah, zgrajenih s temi ogrodji.

Rad bi se zahvalil vsem, ki so me podpirali v času študija, in Adi, ki me je prenašala v času pisanja diplomske naloge. Zahvaljujem se tudi vsem, ki so sodelovali v uporabniškem testu aplikacij, še posebej pa se zahvaljujem svojemu mentorju doc. dr. Alešu Smrdelu za strokovno pomoč in vodenje pri izdelavi diplomske naloge.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	Application Programming Interface	aplikacijski programski vmesnik
XML	Extensible Markup Language	razširljivi označevalni jezik
HTML	Hyper Text Markup Language	jezik za označevanje nadbesedila
CSS	Cascading Style Sheets	kaskadne stilske podloge
SQL	Structured Query Language	strukturirani poizvedovalni jezik
REST	Representational State Transfer	predstavitveni prenos stanja
WCF	Windows Communication Foundation	komunikacijski temelj Windows
UWP	Universal Windows Platform	univerzalna windows platforma
WPF	Windows Presentation Foundation	predstavitveni temelj Windows
GTK#	Gimp Toolkit Sharp	skupek orodij za razvoj uporabniških vmesnikov
MVVM	Model–View–ViewModel	model-pogled-pogledni model
SDK	Software Development Kit	skupek razvojnih orodij
NDK	Native Development Kit	skupek orodij za razvoj Android aplikacij v jeziku C/C++

UI	User Interface	uporabniški vmesnik
XAML	Extensible Application Markup Language	razširljiv aplikacijski označevalni jezik
ORM	Object-Relational Mapping	objektno-relacijsko mapiranje
LINQ	Language Integrated Query	jezikovno integrirana poizvedba
JSON	JavaScript Object Notation	zapis objekta v jeziku JavaScript
PHP	Hypertext Preprocessor	strežniški skriptni programski jezik
CLI	Command Line Interface	vmesnik z ukazno vrstico
MVP	Model-View-Presenter	model-pogled-voditelj

Povzetek

Naslov: Primerjava ogrodij za hkraten navzkrižni razvoj mobilnih aplikacij za več platform

Avtor: Blaž Berglez

Zaradi vse zmogljivejših mobilnih naprav so te postale oblika računalnika, ki ga najpogosteje uporabljamo. S tem razlogom je tudi razvoj mobilnih aplikacij vse pomembnejši. Zaradi razdeljenosti trga pa je, da dosežemo čim večje število uporabnikov, potreben razvoj aplikacije za vsaj dve platformi (iOS in Android). Poleg razvoja aplikacij za vsako platformo posebej pa obstajajo tudi načini za hitrejši (hkraten) razvoj za več platform. V sklopu diplomskega dela so bili najprej raziskani možni načini razvoja mobilnih aplikacij. Nato so bila izbrana tri ogrodja za hkraten navzkrižni razvoj mobilnih aplikacij za več platform, s katerimi je bila razvita enaka enostavna aplikacija. Postopek izdelave aplikacije z vsakim izmed ogrodij (Xamarin, NativeScript in Flutter) je bil tudi podrobno opisan. Na koncu sta bila izvedena še uporabniški test razvitih aplikacij in primerjava razvoja. Najboljše rezultate smo, tako s stališča razvijalca kot tudi testnih uporabnikov, dosegli z ogrodjem Flutter.

Ključne besede: mobilna aplikacija, navzkrižni razvoj, Xamarin, NativeScript, Flutter.

Abstract

Title: Comparison of frameworks for concurrent cross-development of mobile applications for multiple platforms

Author: Blaž Berglez

Due to the mobile devices becoming more and more powerful, these devices have become a form of computers, that we most often use. This is the reason why the development of mobile applications is becoming more and more important. To reach as many users as possible, the development of applications for at least two platforms (iOS and Android) is necessary due to the divisiveness of the market. In addition to individual development of applications for each platform, there are also ways for a faster (concurrent) development for multiple platforms. In the course of the diploma thesis, the possibilities for developing mobile applications were first studied. Next three frameworks for concurrent cross-development of mobile applications for multiple platforms were selected, with which the same simple application was developed. The application development process using each of the selected frameworks (Xamarin, NativeScript, and Flutter) was also described in detail. Finally, a user testing of the developed applications and a comparison of the development were performed. We achieved the best results from both the developer's and test users' point of view with the Flutter framework.

Keywords: mobile application, Cross-platform development, Xamarin, NativeScript, Flutter.

Poglavje 1

Uvod

V zadnjih letih so pametni telefoni in tablični računalniki postali naprave, s katerimi najpogosteje dostopamo do svetovnega spleta [8]. Vse te mobilne naprave uporabljamo za zajem in deljenje fotografij ter videoposnetkov, na njih pregledujemo novice, elektronsko pošto in vse mogoče multimedijske vsebine, uporabljamo jih kot predvajalnike glasbe, na njih igramo igre, z njimi dostopamo do spletnih bank, poleg tega pa jih uporabljamo tudi za dostopanje do najrazličnejših storitve za komunikacijo, ki daleč presegajo samo klice in sporočila, čemur so bili mobilni telefoni na začetku namenjeni. Razlog za takšno popularnost je preprost: pametni telefoni so naprave, ki jih imamo vedno pri roki, njihova zmogljivost pa je blizu zmogljivosti nekaj let starega namiznega računalnika. Osebni računalniki postajajo naprave, ki jih uporabljajo bolj ali manj samo še profesionalni uporabniki in "igračarji", ostali uporabniki pa v glavnem uporabljajo samo še mobilne naprave. Zaradi tega je za ponudnike vsebin in storitev vse bolj pomembno, da so njihove vsebine in storitve dosegljive končnim uporabnikom na mobilnih napravah.

S tem razlogom potreba po razvoju mobilnih aplikacij vztrajno narašča, s povečevanjem števila dosegljivih aplikacij pa narašča tudi zahtevnost uporabnikov. Cilj razvijalcev je izdelati čim kvalitetnejše aplikacije dosegljive čim večjemu številu uporabnikov. Težava pa je, da obstaja več različnih mobilnih platform in če želimo aplikacijo ponuditi vsem uporabnikom, jo moramo

razviti za vse platforme. Najbolj razširjeni mobilni plaformi sta trenutno Android in iOS, zato je najpomembnejše, da aplikacijo razvijemo vsaj za ti dve platformi.

Razvoj aplikacije za več platform je možen na več različnih načinov. Tako lahko pri razvoju uporabljamo pristop z razvojem aplikacije za točno določeno platformo, pristop z razvojem spletne aplikacije, pristop z razvojem hibridne aplikacije, lahko pa tudi uporabimo pristop z uporabo ogrodij za hkraten navzkrižni razvoj aplikacij za več platform. Namen te diplome je pregled in primerjava možnih načinov implementacije aplikacije za različne platforme, razvoj aplikacije v treh izbranih ogrodijh za hkraten navzkrižni razvoj aplikacij (Xamarin, NativeScript in Flutter) ter primerjava postopkov razvoja in razvitih aplikacij. Motivacija za izdelavo takšne primerjave pa izhaja tudi iz lastnih interesov, saj že nekaj let za razvoj mobilnih aplikacij uporabljam ogrodje Xamarin, zato me je tudi zanimalo, kako se primerja z drugimi ogrodji.

V nadaljevanju naprej sledi pregled mobilnih operacijskih sistemov in podrobnejši opis možnih pristopov k razvoju mobilnih aplikacij. V tretjem poglavju je opisana aplikacija, ki je bila v okviru te diplomske naloge razvita z izbranimi ogrodji. V četrtem poglavju je opisano ogrodje Xamarin in postopek razvoja aplikacije z njim. Sledi peto poglavje, kjer je opisano ogrodje NativeScript in postopek razvoja aplikacije z njegovo uporabo. V šestem poglavju se nahaja opis ogrodja Flutter in njegove uporabe za razvoj aplikacije. V sedmem poglavju so zapisane ugotovitve in primerjave aplikacij in ogrodij, v zadnjem, osmem poglavju pa sledi še zaključek.

Poglavje 2

Ozadje

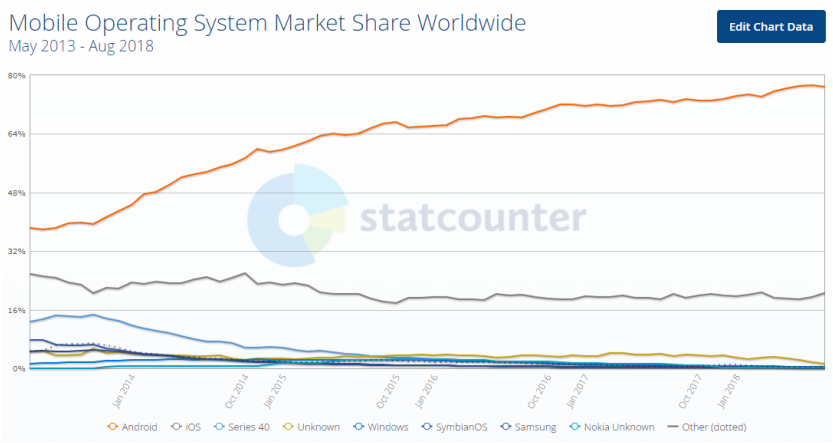
V tem poglavju sta opisana ključna mobilna operacijska sistema, poleg tega pa so opisani tudi možni pristopi k razvoju mobilnih aplikacij. Na koncu je podana še utemeljitev izbire ogrodij, ki so bila v tej nalogi izbrana za hkraten navzkrižni razvoj mobilne aplikacije za več platform.

2.1 Mobilni operacijski sistemi

O razvoju mobilnih aplikacij težko govorimo, ne da bi se dotaknili samih mobilnih operacijskih sistemov, na katerih aplikacije tečejo. Najbolj razširjena operacijska sistema sta iOS in Android, ki skupaj zajemata kar okoli 97 odstotkov svetovnega tržnega deleža, kar predstavlja približno 2,5 milijarde naprav. Na tretjem mestu je mobilni operacijski sistem Windows, katerega tržni delež je od leta 2014 iz slabih 4 odstokov padel na 0,4 odstokov (slika 2.1).

2.1.1 iOS

iOS je mobilni operacijski sistem zasnovan in razvit v podjetju Apple Inc., namenjen pametnim telefonom (iPhone), tabličnim računalnikom (iPad) in predvajalnikom multimedisjkih vsebin (iPod Touch) podjetja Apple. Na začetku ga je Steve Jobs imenoval kar OS X, ker je bil osnovan na podobnem



Slika 2.1: Svetovni tržni delež mobilnih operacijskih sistemov v obdobju 2013-2018 [3].

jedru Unix kot Applov operacijski sistem, namenjen namiznim računalnikom. Ko pa je 29. junija 2007 na trg prišel prvi iPhone, se je tudi operacijski sistem preimenoval v iPhone OS. Prvi iPhone je revolucioniral trg pametnih telefonov tudi po zaslugi iPhone OS, ki je ponudil enostavno uporabo funkcionalnosti predvajalnika glasbe, telefona, žepnega fotoaparata in spletnega brskalnika v eni napravi z zaslonom na dotik. Ob izidu iPhone 3G 11. julija leta 2008 je operacijski sistem dosegel nov mejnik. Nova različica je ponudila možnost nalaganja aplikacij zunanjih razvijalcev iz spletne trgovine za aplikacije, ki se danes imenuje AppStore. Ob izidu četrte različice se je iPhone OS preimenoval v iOS, saj ni bil več nameščen samo na iPhonih ampak tudi na ostalih iNapravah – iPad in iPod Touch. Zadnja različica (iOS 12.0) je izšla 17. septembra 2018. Operacijski sistem iOS obvladuje okoli 20-odstotni svetovni tržni delež mobilnih operacijskih sistemov, v nekaterih predelih sveta (Združene države Amerike, Kanada, Združeno Kraljestvo) pa je njegov tržni delež okrog 50 odstotkov [2]. Aplikacije so dostopne v Applovi trgovini za aplikacije iOS – AppStore, kjer je na voljo preko 3 milijone aplikacij in iger [5].

2.1.2 Android

Android je operacijski sistem, osnovan na linuxovem jedru. Namenjen je predvsem mobilnim napravam z zaslonom na dotik, kot so pametni telefoni in tablični računalniki. Njegov razvoj se je začel leta 2003 v podjetju Android inc., katerega soustanovitelj je bil tudi Andy Rubin, po katerem je Android dobil ime. Sprva je bil njegov namen ponuditi boljši operacijski sistem, namenjen digitalnim kameram, ki bi omogočal povezovanje kamere preko brezžične povezave z osebnimi računalniki in Android oblakom. Ker pa je uporaba digitalnih kamer začela upadati, se je podjetje preusmerilo v izdelavo operacijskega sistema, namenjenega mobilnim telefonom. Leta 2005 je podjetje odkupil Google in leta 2007 skupaj s številnimi proizvajalci mobilnih naprav in ponudniki mobilnih storitev, kot so HTC, LG, Samsung, T-Mobile in Sprint, ustanovil organizacijo Open Handset Alliance (OHA). Android je 5. novembra 2007, ko je izšla prva beta različica, namenjena razvijalcem, postal prostodostopna platforma. Prva komercialno dostopna naprava z nameščenim operacijskim sistemom Android je bil pametni telefon HTC Dream, ki je prišel na trg seprembra 2008. Do konca leta 2009 je izšla že peta različica operacijskega sistema (2.0 – Eclair), nato se je vzorec izhajanja ustalil z izidom ene nove različice vsako leto, zadnja različica (9.0 – Pie) je izšla 6. avgusta 2018. Danes Android obvladuje okoli 76-odstotni tržni delež mobilnih operacijskih sistemov, uporablja pa se tudi za druge naprave, kot so TV-sprejemniki, avtomobili, pametne ure in miniaturni računalniki, ki so po navadi v obliki malo večjih USB-ključev s HDMI-izhodom in jih lahko uporabimo kot igralne konzole ali TV-škatle (ang. TV Box). Android omogoča nalaganje aplikacij neposredno z odpiranjem .apk datotek, dostopne pa so tudi v različnih trgovinah za aplikacije Android, kot so SlideMe, Amazon Appstore in 1Mobile Market, največja med njimi pa je Google Play, kjer se nahaja okrog 3,3 milijonov aplikacij [4].

2.2 Razvoj mobilnih aplikacij

Pristope k razvoju mobilnih aplikacij lahko razvrstimo v štiri kategorije:

- razvoj platformno specifičnih aplikacij,
- razvoj spletnih aplikacij,
- razvoj hibridnih aplikacij,
- razvoj z uporabo ogrodij za navzkrižni razvoj za več platform.

Odločitev, kateri pristop bomo izbrali za razvoj, je odvisna predvsem od tega, katere platforme želimo podpirati, vrste aplikacije, ki jo razvijamo, količine razpoložljivih sredstev, končnih zahtev aplikacije, neposredne odvisnosti aplikacije od posamezne platforme, količine pričakovanega vzdrževanja in nadgrajevanja aplikacije in pa seveda tudi od predhodnega znanja razvijalca oziroma splošne usmerjenosti organizacije, ki aplikacijo razvija. Večina razvijalcev komercialnih aplikacij želi ponuditi svoj produkt čim večjemu številu uporabnikov. To se trenutno lahko doseže z razvojem aplikacije za platformi iOS in Android, ki skupaj zajemata kar okoli 97 odstotkov tržnega deleža mobilnih operacijskih sistemov.

2.2.1 Razvoj platformno specifičnih aplikacij

Razvoj platformno specifičnih aplikacij pomeni ločen razvoj aplikacije za vsako platformo, na kateri želimo, da se naša aplikacija izvaja.

Za razvoj aplikacij iOS je na voljo uradno Applovo razvojno okolje XCode. Namestimo ga lahko samo na Applove računalnike z operacijskim sistemom OS X. Aplikacije razvijamo v programskem jeziku Objective-C ali Swift, ki ponuja hitrejši razvoj v primerjavi z Objective-C in ga počasi zamenjuje. Uporabniški vmesnik lahko gradimo z uporabo grafičnega urejevalnika ali z urejanjem datotek XML imenovanih Storyboard.

Za razvoj aplikacij Android je na voljo vtičnik za razvojna okolja Eclipse in NetBeans IDE, najpogosteje pa razvijalci uporabljajo Android studio. An-

droid studio temelji na razvojnem okolju podjetja IntelliJ IDEA, je prirejeno za razvoj aplikacij Android in je tudi uradno priporočeno. Dostopno je brezplačno za operacijske sisteme Windows, Linux in OS X. Aplikacije Android razvijamo v programskem jeziku Java ali Kotlin, ki je modernejši, hitrejši za razvoj in naj bi v prihodnosti nadomestil Javo za razvoj aplikacij za sistem Android. Uporabniški vmesnik definiramo z uporabo datotek XML s končnico .axml ali z uporabo grafičnega urejevalnika.

Razvoj platformno specifičnih aplikacij omogoča neposreden dostop do najnovejše različice aplikacijskega programskega vmesnika (API) posameznega operacijskega sistema in vseh domorodnih knjižnic in razširitev. Rezultat je domorodna aplikacija, ki izmed vseh pristopov ob pravilni uporabi tehnologij zagotavlja največjo učinkovitost delovanja aplikacije. Za razvoj platformno specifičnih aplikacij se najpogosteje odločajo razvijalci, ki niso omejeni s sredstvi, imajo dovolj predznanja oz. razvijalcev za razvoj dveh ločenih aplikacij, razvijajo samo za eno platformo ali pa je zelo pomembna učinkovitost delovanja aplikacije.

2.2.2 Razvoj spletnih aplikacij

Najpreprostejši način, da naredimo našo aplikacijo dosegljivo vsem uporabnikom, ne glede na to, kakšne vrste mobilne naprave oziroma platforme uporabljajo, je razvoj za mobilne naprave prirejene spletne aplikacije. Do spletnih aplikacij lahko uporabniki dostopajo z uporabo spletnega brskalnika. Za razvoj mobilnih spletnih aplikacij lahko uporabimo standardne spletne tehnologije, kot so HTML(5), CSS(3) in JavaScript. Na strani strežnika pa se pogosto uporablja katero izmed ogrodij za spletne aplikacije. Najpogosteje uporabljeni programski jeziki v spletnih ogrodjih so:

- Java (JavaServer Faces, Struts, Hibernate, Google Web Toolkit ...),
- PHP (Laravel, Symfony, CodeIgniter, Yii 2 ...),
- Python (Django, Pyramid, TurboGears, Web2py, Microframeworks ...),

- JavaScript (Angular(JS), React, Ember.js, Vue.js, Backbone.js ...).

Prednost spletnih aplikacij je, da so dostopne na mobilnih napravah, ne da bi jih bilo treba namestiti, razvijalci pa imajo poenostavljeno vzdrževanje in nadgrajevanje, saj je treba skrbeti samo za eno aplikacijo, ki teče na njihovih strežnikih. Slabost pa je, da uporabnik za uporabo nujno potrebuje internetno povezavo, poznati mora spletni naslov aplikacije oz. si ga mora shraniti v zaznamek v spletnem brskalniku, aplikacija pa nima dostopa do funkcionalnosti operacijskega sistema. Razvoj spletne aplikacije je po navadi prvi korak obstoječih in novih ponudnikov spletnih storitev. Primeren je predvsem za aplikacije, ki ne uporabljajo funkcionalnosti naprave, na kateri tečejo, in aplikacije, ki lahko imajo sicer veliko število različnih uporabnikov, vendar posamezni uporabniki ne uporabljajo aplikacije zelo pogosto.

2.2.3 Razvoj hibridnih aplikacij

Hibridne aplikacije so v osnovi spletne aplikacije, prirejene za mobilne naprave, zapakirane v domorodni spletni pogled (ang. Web View) na posamezni platformi. Domorodni spletni pogled poskrbi za prikaz in izvajanje aplikacije, zgrajene s spletnimi tehnologijami, kot so HTML, CSS in JavaScript, ali katerim izmed spletnih ogrodij, ki za izvajanje ne potrebujejo strežnika. V poštev pridejo predvsem spletna ogrodja, zgrajena v programskem jeziku JavaScript. Za razvoj hibridnih aplikacij lahko uporabimo enega izmed številnih za to namenjenih ogrodij. Ta ogrodja omogočajo tudi omejen dostop do funkcionalnosti programskega vmesnika operacijskega sistema. Za dostop do domorodnih knjižnic in funkcionalnosti posamezne platforme, ki jih ogrodje ne vsebuje, je potreben razvoj vtičnika, namenjenega ogrodju, ki ga uporabljamo. Hibridne aplikacije po navadi ne dajejo občutka domorodne aplikacije, učinkovitost delovanja pa je odvisna od uporabljenega ogrodja, na splošno pa je slabša tako od platformno specifično razvitih aplikacij, kot tudi aplikacij razvitih z uporabo ogrodij za hkraten navzkrižni razvoj.

Prednost hibridnega razvoja je najhitrejši razvoj in enoten izgled aplikacije, ne glede na platformo, čeprav lahko pride do razlik na različnih različicah

operacijskega sistema, podobno kot so opazne razlike v izvajanju spletnih strani v različnih spletnih brskalnikih. Za razvoj hibridne aplikacije se najraje odločajo spletni razvijalci ter tisti razvijalci, ki se ne želijo ukvarjati s posebnostmi posamezne platforme in jim učinkovitost delovanja aplikacije ne predstavlja najvišje prioritete.

V nadaljevanju je na kratko opisanih nekaj najpogosteje uporabljenih ogrodij, namenjenih razvoju hibridnih mobilnih aplikacij.

Apache Cordova

Apache Cordova je ogrodje, ki omogoča uporabo standardnih spletnih tehnologij – HTML5, CSS3 in JavaScript za razvoj hibridnih aplikacij. Spletne aplikacije zapakira v paket, namenjen izvajanju na platformi, določeni ob izgradnji aplikacije in preko vgrajenih ter po meri zgrajenih vtičnikov omogoča dostop do sistemskih programskih vmesnikov in domorodnih knjižnic. Je osnova, na kateri so zgrajena številna ostala ogrodja.

Adobe PhoneGap

PhoneGap in Apache Cordova sta bila na začetku isti produkt. Sedaj pa je PhoneGap ločeno ogrodje, ki za delovanje uporablja Apache Cordovo in poenostavi uporabo z uporabo grafičnega vmesnika namesto ukazne vrstice za izgradnjo aplikacij in hitrejšim zagonom aplikacij za testiranje.

Ionic Framework

Tudi Ionic za delovanje uporablja Apache Cordovo vendar pa uporabnika usmerja v uporabo zelo priljubljenega spletnega aplikacijskega ogrodja Angular. Omogoča uporabo JavaScript in TypeScript in je trenutno najbolj priljubljeno ogrodje za razvoj hibridnih aplikacij [1].

Framework 7

Podobno kot Ionic tudi Framework 7 za delovanje uporablja Apache Cordovo, prinaša pa lastno aplikacijsko ogrodje, zgrajeno na Vue.js, v katerega so dodatno vgrajeni predpripravljeni gradniki, ki replicirajo izgled domorodnih

gradnikov platforme iOS in Android.

Xamarin

Sam Xamarin sicer ni ogrodje, namenjeno razvoju hibridnih aplikacij, vendar ga je na ta način vseeno mogoče uporabiti. Potrebna je implementacija hibridnega spletnega pogleda, ki omogoča registracijo metod, na katere se je mogoče sklicevati iz kode JavaScript spletne aplikacije, ki jo je ob zagonu treba naložiti v hibridni pogled.

2.2.4 Razvoj z uporabo ogrodij za navzkrižni razvoj za več platform

Ogrodja za hkraten navzkrižni razvoj za več platform omogočajo razvoj aplikacije za različne platforme z uporabo enega programskega jezika in enega razvojnega okolja. Z uporabo programskega jezika in načina razvoja, ki ga določa ogrodje, razvijalci razvijejo aplikacije, ki jih zna ogrodje prevesti v platformno specifične pakete oz. v kodo v zbirnem jeziku, namenjeno izvajanju v določenem izvajalnem okolju. Proces je podoben kot pri razvoju Android aplikacije v programskem jeziku Kotlin, kjer se aplikacija prevede najprej v javansko zložno kodo in nato v Dalvik zložno kodo, ki se potem izvaja preko Android izvajalnega okolja.

Razlika je v tem, da se aplikacije, zgrajene z uporabo ogrodij za navzkrižni razvoj, ne izvajajo v izvajalnem okolju platforme, ampak v izvajalnem okolju, ki omogoča izvajanje prevedene kode programskega jezika, ki ga za razvoj določa ogrodje. Izvajalno okolje teče neposredno na operacijskem sistemu in se v primeru Androida izvaja vzporedno z Android izvajalnim okoljem. Ker pa potrebno izvajalno okolje ponavadi ni prednameščeno na operacijskem sistemu, ga ogrodje za razvoj ob izgradnji zapakira skupaj s samo aplikacijo in ga po potrebi namesti ob prvem zagonu aplikacije. Zato so namestitveni paketi na ta način zgrajene aplikacije nekoliko večji od paketov zgrajenih s platformno specifičnim razvojem aplikacije. Zelo pomemben element izvajalnega okolja pa je tudi tako imenovani most, ki služi kot povezava med upra-

vljano in neupravljano kodo oz. omogoča uporabo sistemskega programskega vmesnika z uporabo programskega jezika, ki platformi ni domoroden. Aplikacije, razvite z ogrodji za navzkrižni razvoj za več platform, lahko (odvisno od ogrodja) za izris uporabniškega vmesnika v končni aplikaciji uporabljajo domorodne grafične elemente in se s tem po izgledu in občutku zelo približajo platformno specifično razvitim aplikacijam. Tudi učinkovitost delovanja na ta način razvitih aplikacij je zelo blizu platformno specifično razvitim aplikacijam. Dostopnost najnovejših sistemskih programskih vmesnikov posamezne platforme in uporaba domorodnih knjižnic je odvisna od ogrodja.

Čeprav razvoj s temi ogrodji poteka v skupnem programskem jeziku pa je priporočljivo, da razvijalci poznajo vsaj osnove platformno specifičnega razvoja za implementacijo morebitnih platformno specifičnih funkcionalnosti. S tem pristopom se strošek, čas razvoja in obseg vzdrževanja v primerjavi z razvojem platformno specifičnih aplikacij občutno zmanjša. Za uporabo ogrodij za navzkrižni razvoj za več platform se najraje odločijo razvijalci, ki bi radi v čim krajšem času uporabnikom ponudili kvalitetno aplikacijo, poznajo način programiranja v izbranem ogrodju in nimajo dovolj razvijalcev, sredstev ali znanja, da bi razvijali platformno specifične aplikacije.

V to kategorijo poleg ostalih spadajo tudi ogrodja, ki so bila v tej nalogi uporabljena za razvoj aplikacij. Izbrana ogrodja so podrobneje opisana v naslednjih poglavjih, tu pa sledi kratek opis ogrodij, ki niso bila izbrana.

React Native

React Native je prostodostopno ogrodje, ki omogoča uporabo prirejenega spletnega aplikacijskega ogrodja React za hkratni navzkrižni razvoj aplikacij za platformi iOS in Android z uporabo programskega jezika JavaScript. Razvilo ga je podjetje Facebook in je eno izmed najpriljubljeneših ogrodij za hkraten navzkrižni razvoj mobilnih aplikacij za več platform. Razlog za njegovo priljubljenost je predvsem hiter in razvoju s spletnim aplikacijskim ogrodjem React podoben razvoj aplikacij.

Appcelerator Titanium

Ogrodje Appcelerator je prvič izšlo leta 2008 in je bilo sprva namenjeno navzkrižnemu razvoju aplikacij za namizne računalnike. Leta 2009 je izšla različica, ki je omogočala tudi razvoj aplikacij iOS in Android. Za razvoj aplikacij z Appceleratorjem uporabljamo programski jezik JavaScript. JavaScript je uporabljen tudi za gradnjo uporabniškega vmesnika, ki pa ga aplikacija, zgrajena z Appceleratorjem, izrisuje z uporabo domorodnih elementov.

Weex

Weex je še eno ogrodje, ki za skupni programski jezik uporablja programski jezik JavaScript. Omogoča razvoj aplikacij Android, iOS in Web in uporabo več aplikacijskih ogrodij, najbolje pa sta podprta Vue.js in Rax.

Jasonette

Jasonette se od ostalih ogrodij razlikuje po tem, da aplikacije, razvite z njim, delujejo podobno kot spletni brskalnik. Prikazan uporabniški vmesnik zgradi iz domorodnih elementov platforme na podlagi podatkov v obliki JSON, ki jih pridobi s spletnega strežnika. Logika aplikacije se tako nahaja na spletnem strežniku. Jasonette je primeren predvsem za aplikacije s statičnimi zasloni.

2.3 Izbrana ogrodja

Za podrobnejšo analizo in razvoj aplikacije v tej diplomski nalogi so bila izbrana ogrodja Xamarin, NativeScript in Flutter. Xamarin je bil izbran, ker je predstavlja osnovo za primerjavo z ostalima dvema ogrodjama. NativeScript je bil izbran, ker omogoča uporabo Angular spletnega aplikacijskega ogrodja. Angular je odlično ogrodje za razvoj spletnih aplikacij, s katerim tudi že imam nekaj iskušenj. Flutter še niti ni dosegel prve različice, na spletu pa vzbuja veliko zanimanja vse od kar ga je Google najavil leta 2017.

Poglavje 3

Specifikacije aplikacije

Aplikacija, ki je bila v tej nalogi razvita za namen primerjave ogrodij za hkraten navzkrižni razvoj mobilnih apikacij za več platform, je bila določena pred začetkom razvoja. Zastavljena je tako, da pokriva osnovne funkcionalnosti povprečne mobilne aplikacije. Poimenovana je bila Notes App, njen osnovni namen pa je shranjevanje, urejanje in pregledovanje zapiskov z možnostjo nastavitve obvestila za posamezen zapisek.

3.1 Funkcionalnosti in izgled aplikacije

Aplikacija omogoča registracijo in prijavo registriranih uporabnikov. Prijavljeni uporabniki imajo možnost dodajanja zapiskov, ki jim lahko kot priponke dodajo tudi slike in jim nastavijo obvestilo. Zapiske lahko pregledujejo, urejajo in brišejo. Osnovne funkcionalnosti aplikacije, ki so bile uporabljene za primerjavo ogrodij, so:

- shranjevanje podatkov v lokalno podatkovno bazo SQLite,
- uporaba spletne storitve REST za prijavo, registracijo in sinhronizacijo zapiskov s spletnim strežnikom,
- varno shranjevanje podatkov za shranjevanje avtentikacijskega žetona uporabnika,

- izbira slik iz naprave,
- prikaz lokalnih obvestil.

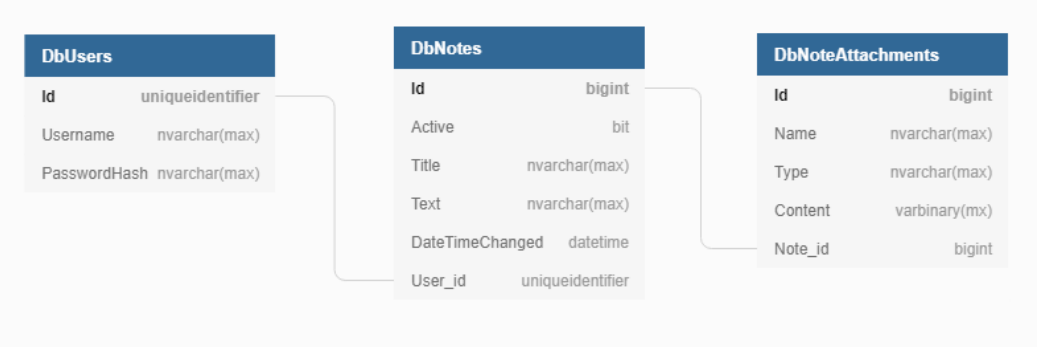
Pred samim razvojem je bila določena tudi grafična podoba aplikacije in okvirna barvna shema. Osnovna barva aplikacije je temno siva z oranžnim poudarkom. Aplikacija je sestavljena iz petih zaslonov:

- zaslona za prijavo,
- zaslona za registracijo,
- zaslona, ki prijazuje seznam shranjenih zapiskov,
- zaslona za dodajanje in urejanje zapiska,
- zaslona za izbiro datuma in ure obvestila.

Zaslon za prijavo vsebuje vnosni polji za uporabniško ime in geslo, gumb za prijavo in gumb za registracijo, ki sproži navigacijo na zaslon za registracijo. Zaslon za registracijo je sestavljen iz vnosnih polj za uporabniško ime, geslo in potrditev gesla, gumba za registracijo, ki pa se izvede, če se gesli ujemata in uporabniško ime še ni zasedeno. Na zaslonu za registracijo se nahaja tudi navigacijska vrstica, ki omogoča navigacijo nazaj na zaslon za prijavo. Zaslon s seznamom zapiskov omogoča tudi brisanje zapiskov, na njem pa se nahaja tudi plavajoči gumb, s katerim je mogoče dodajati zapiske, in navigacijska vrstica, ki omogoča odpiranje navigacijskega predala. Navigacijski predal se odpira iz leve strani, na njem pa se nahaja informacija o prijavljenem uporabniku in menijske opcije za dodajanje zapiska, sinhronizacjo s strežnikom in odjavo uporabnika. Zaslon za dodajanje ali urejanje zapiska vsebuje vnosni polji za naslov ter besedilo zapiska, pod njima pa je gumb za dodajanje priponke, ki odpre galerijo za izbiro slike ter seznam priponk. V navigacijski vrstici zaslona se na desni strani nahaja dodaten gumb, ki odpre zaslon za izbiro datuma in ure obvestila. Na zaslonu za izbiro datuma in ure sta izbirnika za datum in čas ter gumb za potrditev, navigacijska vrstica pa omogoča preklic izbire datuma in časa obvestila. Obvestilo se nastavi šele po potrditvi shranjevanja zapiska.

3.2 Opis spletne storitve

Spletna storitev REST, ki jo aplikacija uporablja za registracijo in prijavo uporabnikov ter centralno shranjevanje zapiskov, je bila razvita z uporabo tehnologije .NET WCF in poimenovana Notes.Service. Za shranjevanje podatkov uporablja podatkovno bazo SQL. Za branje in pisanje podatkov v podatkovno bazo uporablja ogrodje EntityFramework po pristopu Code-First, ki omogoča definicijo tabel podatkovne baze z uporabo razredov v jeziku C# in uporabo LINQ poizvedb za branje podatkov po metodi ORM. Shema podatkovne baze je prikazana na sliki 3.1. Gesla uporabnikov so v podat-



Slika 3.1: Shema podatkovne baze storitve Notes.Service.

kovni bazi shranjena v obliki zgoščene vrednosti s pripeto soljo, pretvorjena v base64 znakovni niz.

Storitev zajema štiri metode:

- ***Ping***: metoda tipa GET, namenjena preverjanju dosegljivosti strežnika.

- ***RegisterNewUser***: metoda tipa POST, namenjena registraciji uporabnika. Sprejme objekt uporabnika v obliki JSON, ki vsebuje uporabniško ime in geslo. Metoda najprej preveri, če uporabniško ime še ni zasedeno, nata izračuna zgoščeno vrednost gesla, shrani novega upo-

rabnika v podatkovno bazo in vrne rezultat.

- ***Login***: metoda tipa POST, namenjena prijavi uporabnika. Sprejme objekt uporabnika v obliki JSON in preveri, če se izračunana zgoščena vrednost gesla ujema z geslom shranjenim v podatkovni bazi. Če se, vrne avtentikacijski žeton uporabnika, ki je ID uporabnika, drugače vrne napako ob avtentikaciji.

- ***Synchronize***: metoda tipa POST, namenjena sinhronizaciji zapiskov. Metoda zahteva uporabo avtentikacijskega žetona in sprejme polje zapiskov, ki so bili na odjemalcu (aplikaciji) spremenjeni, odstranjeni ali dodani in čas zadnje sinhronizacije. Zapisek se v podatkovni bazi posodobi samo v primeru, ko je poslan datum in čas spremembe zapiska večji od trenutno shranjenega datuma in časa zadnje spremembe zapiska. Na ta način so v centralni podatkovni bazi vedno shranjene zadnje različice zapiskov, ne glede na to na kateri napravi so bili zapiski spremenjeni. Rezultat metode je polje vseh zapiskov, dodanih, spremenjenih ali izbrisanih oz. deaktiviranih po podanem času zadnje sinhronizacije, in trenutni čas. Vrnjene zapiske in čas zadnje sinhronizacije mora odjemalec shraniti v svojo podatkovno bazo.

Poglavje 4

Xamarin

Xamarin je produkt podjetja Xamarin, ki so ga maja leta 2011 ustanovili razvijalci prostodostopnih projektov Mono, Mono for Android in MonoTouch. Mono je produkt, ki omogoča izvajanje z ogrodjem .NET zgrajenih programov na operacijskih sistemih Linux. Mono for Android in MonoTouch sta osnovi produktov Xamarin.Android in Xamarin.iOS in omogočata izvajanje z ogrodjem .NET zgrajenih aplikacij na operacijskih sistemih Android in iOS.

Za razvoj aplikacij s Xamarinom uporabljamo programski jezik C#. Jezik C# je visokonivojski, splošno namenski, močno tipiziran, dogodkovno voden in objektno orientiran programski jezik, ki ga je okrog leta 2000 razvilo podjetje Microsoft.

Prva različica Xamarina, ki je izšla leta 2011, je omogočala hkraten navzkrižni razvoj aplikacij za platformi Android in iOS z možnostjo deljenja kode poslovne logike in ločenim razvojem uporabniškega vmesnika za vsako platformo posebej. Tak način razvoja sedaj imenujemo razvoj po klasični metodi in omogoča nekje med 50 in 80 odstotki deljene kode med platformami (odvisno od količine poslovne logike).

Leta 2014 je izšla nova komponenta Xamarina – Xamarin Forms. Xamarin Forms poleg deljenja kode poslovne logike omogoča tudi navzkrični razvoj uporabniškega vmesnika. Na začetku je podpiral platforme Android,

iOS in Windows Phone, zadnja različica pa podpira platforme Android, iOS, GTK#, Mac, Tizen, UWP in WPF. Z uporabo Xamarin Forms lahko dosežemo do okrog 96 odstotkov med platformami deljene kode. Xamarin Forms je uporabljen tudi za razvoj aplikacije v tej nalogi.

Leta 2016 je Xamarin prevzelo podjetje Microsoft in ga sedaj ponuja kot eno izmed komponent v njihovem razvojnem okolju Visual Studio. Od takrat se je precej izboljšala tudi stabilnost komponent Xamarin.Android in Xamarin.iOS, saj njune funkcionalnosti Microsoft počasi zamenjuje z lastnimi implementacijami.

4.1 Način delovanja

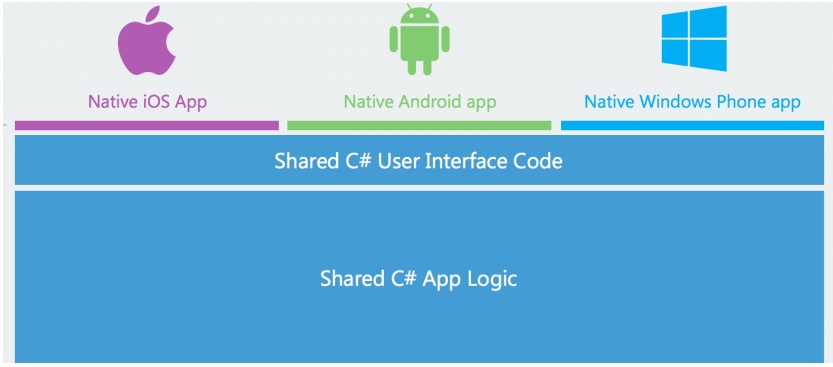
Za izvajanje aplikacij zgrajenih s Xamarinom na platformah iOS in Android skrbita komponenti Xamarin.iOS in Xamarin.Android, katerih osnova je izvajalno okolje Mono.

Zaradi omejitev operacijskega sistema iOS, ki preprečuje spreminjanje kode med izvajanjem, se aplikacija iOS ob izgradnji v celoti prevede predčasno (ang. Ahead of Time compilation). Takšnih omejitev na Androidu ni, zato Xamarin.Android omogoča tako predčasno prevajanje kot tudi prevajanje ravno v času (ang. Just in Time compilation).

Komponenti Xamarin.iOS in Xamarin.Android vsebujeta tudi tako imenovane vezave (ang. Bindings). Vezave so preslikave metod sistemskega programskega vmesnika v obliko, ki jo je mogoče uporabljati s programskim jezikom C#. Če želimo v Xamarinu uporabiti domorodne knjižnice, moramo za ta namen napisati lastno knjižnico vezav (ang. Bindings Library).

4.1.1 Xamarin Forms

Xamarin Forms ni ločen produkt, ampak za delovanje uporablja komponenti Xamarin.iOS in Xamarin.Android. Predstavlja dodatni nivo abstrakcije, ki zajema presek gradnikov uporabniških vmesnikov podprtih platform in tako omogoča definicijo uporabniškega vmesnika na enoten način (slika 4.1). Z



Slika 4.1: Zgradba aplikacije z uporabo Xamarin Forms [6].

uporabo Xamarin Forms se za izris uporabniškega vmesnika na posamezni platformi uporabijo domorodni elementi platforme. Pomembno vlogo imajo pri tem izrisovalniki (ang. Renderers), ki povezujejo abstrakcije gradnikov Xamarin Forms z domorodnimi gradniki na posamezni platformi.

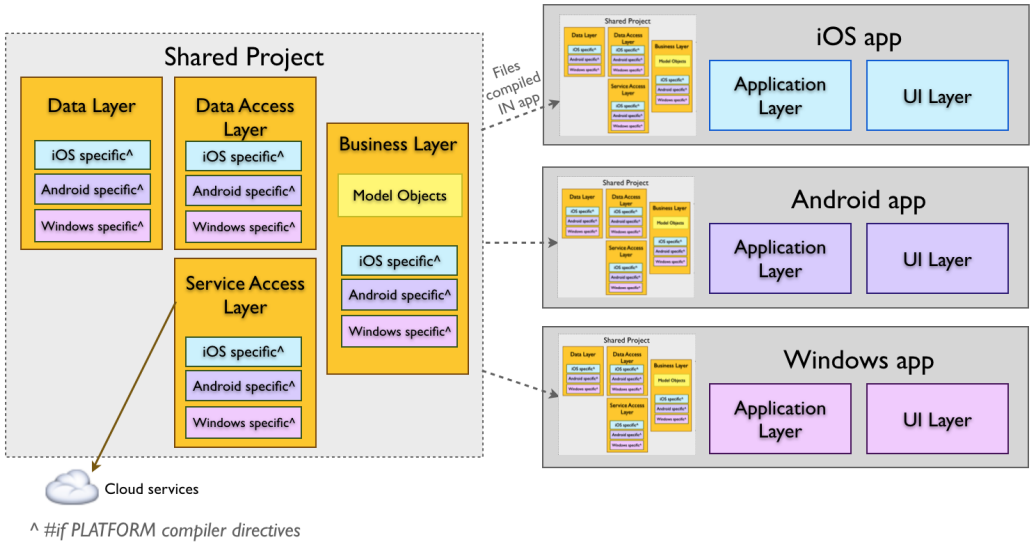
4.2 Arhitektura aplikacije

Za razvoj aplikacij s Xamarinom imamo na voljo dva načina deljenja kode med platformami:

- uporaba deljenih (ang. Shared) projektov (slika 4.2),
- uporaba projektov tipa .NET Standard Library (slika 4.3).

Z uporabo deljenih projektov imamo iz projektov za deljeno kodo neposreden dostop do platformno specifičnih implementacij in funkcij specifičnih za posamezno platformo. Za uporabo platformno specifičnih operacij uporabimo #if direktive prevajalniku, kar pa lahko privede do težko berljive kode.

Knjižnica .NET Standard Library zamenjuje knjižnico Portable Class Library in vsebuje skupne implementacije vseh različic ogrodij .NET (.NET Framework, .NET Core, Mono, Xamarin.iOS, Xamarin.Android, Xamarin.Mac



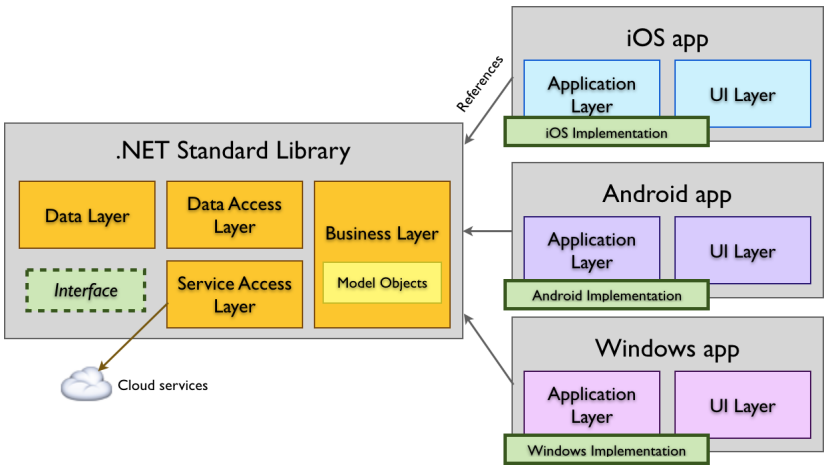
Slika 4.2: Koncept arhitekture aplikacije z uporabo deljenih projektov [7].

in platforme UWP). Z uporabo knjižnic .NET Standard iz projektov s skupno kodo nimamo neposrednega dostopa do platformno specifičnih implementacij, je pa preglednost kode večja in omogoča boljšo razčlenjenost aplikacije v manjše projekte. To strategijo deljenja kode smo uporabili pri razvoju aplikacije v tej nalogi.

4.2.1 MVVM

Okrajšava MVVM predstavlja Model-View-ViewModel in je arhitekturni vzorec, ki ga je razvil Microsoft, da bi poenostavil dogodkovno vodeno programiranje uporabniških vmesnikov. Arhitekturni vzorec MVVM narekuje uporabo treh komponent:

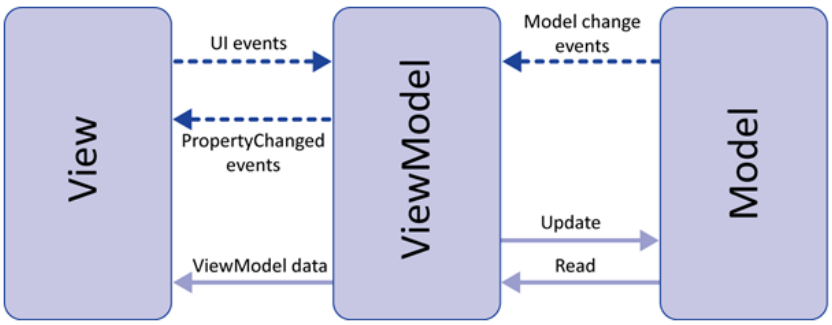
- **model (ang. Model)**, ki predstavlja podatke oz. objekt, ki jih pogled prikazuje ali z njimi operira,
- **pogled (ang. View)**, ki določa izgled uporabniškega vmesnika in je po navadi definiran v označevalnem jeziku XAML,



Slika 4.3: Koncept arhitekture aplikacije z uporabo projektov tipa .NET Standard Library [7].

- **pogledni model (ang. ViewModel)**, ki deluje kot posrednik med pogledom in modelom. Vsebuje podatke, ki jih pogled prikazuje, in metode, ki se izvedejo ob interakciji uporabnika z zaslonom.

Diagram arhitekturnega vzorca MVVM je prikazan na sliki 4.4. Pogled in



Slika 4.4: Diagram arhitekturnega vzorca MVVM [11].

pogledni model komunicirata z uporabo dvosmerne podatkovne vezave (ang. Data Binding). Za osvežitev prikazanih podatkov v pogledu mora pogledni model pogledu sporočiti, da so se podatki spremenili, kar stori s sprožitvijo

dogodka, na katerega pogled posluša (PropertyChanged). Pogledni model nima reference na pogled, zato je lahko razvoj uporabniškega vmesnika povsem ločen od razvoja poslovne logike, lažje pa je tudi testiranje, saj lahko teste enote (ang. Unit Tests) poganjamo kar na poglednem modelu.

4.3 Razvoj aplikacije

Razvoj aplikacij s Xamarinom je potekal izključno na računalniku z nameščenim operacijskim sistemom Windows s povezavo z Applovim računalnikom Mac mini za razhroščevanje in testiranje aplikacije iOS.

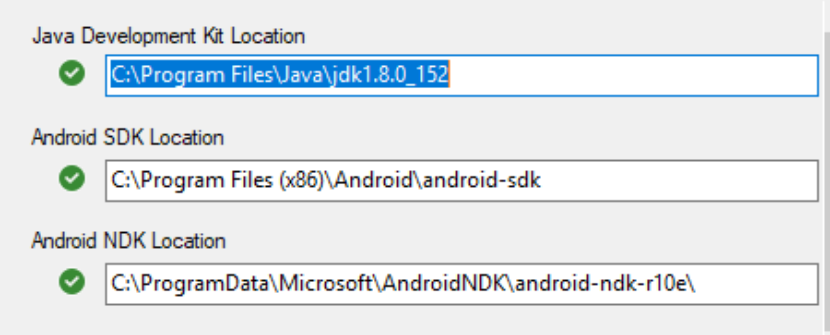
4.3.1 Postavitev delovnega okolja

Windows: Za namestitev Xamarina je potrebna namestitev Microsoftovega integriranega razvojnega okolja Visual Studio. Ob zagonu namestitvenega paketa nam čarovnik ponudi izbiro različnih paketov. Za delo s Xamarinom je treba izbrati paket "Mobile development with .NET" v kategoriji "Mobile & Gaming". Če imamo Visual Studio že nameščen, paket dodamo z uporabo Visual Studio Installerja. V razdelku "Installation details" lahko dodatno modificiramo, katere komponente bodo nameščene. Namestitev paketov "Android SDK" in "Android Emulator" ni potrebna, če sta na računalniku že nameščena.

Mac OS X: Izgradnja aplikacije iOS ni mogoča brez Applovega razvojnega okolja XCode, ki pa ga je mogoče namestiti samo na računalnike z operacijskim sistemom Mac OS X. Xamarin oz. Visual Studio omogoča povezavo z računalnikom Mac preko lokalnega omrežja. Preko vzpostavljene povezave je mogoče aplikacije iOS zgraditi in jih poganjati v simulatorju iOS ali fizični napravi, ki je priključena v računalnik Mac. Da pa lahko takšno povezavo vzpostavimo, je na računalniku Mac treba namestiti najprej razvojno okolje XCode in nato še Microsoftovo razvojno okolje Visual Studio for Mac. Pred namestitvijo je pomembno, da je operacijski sistem posodobljen, saj le najnoveša različica operacijskega sistema podpira namestitve najnovejše

različice XCode, ki pa mora biti usklajena z različico Xamarina. XCode namestimo z uporabo Applovega App Stora. Po namestitvi ga je treba vsaj enkrat zagnati, da se namestitev zaključi. Proces namestitve Visual Studio for Mac je podoben namestivi Visual studia na računalniku Windows. Ob izbiri komponent lahko izključimo namestitev Android komponent, kar pohitri namestitveni proces. Na računalniku Mac je treba vključiti oddaljeno prijavo. To storimo tako, da v sistemskih nastavitvah izberemo opcijo "Sharing" in nato obljukamo možnost "Remote Login" in izberemo uporabnike, ki jim želimo omogočiti dostop.

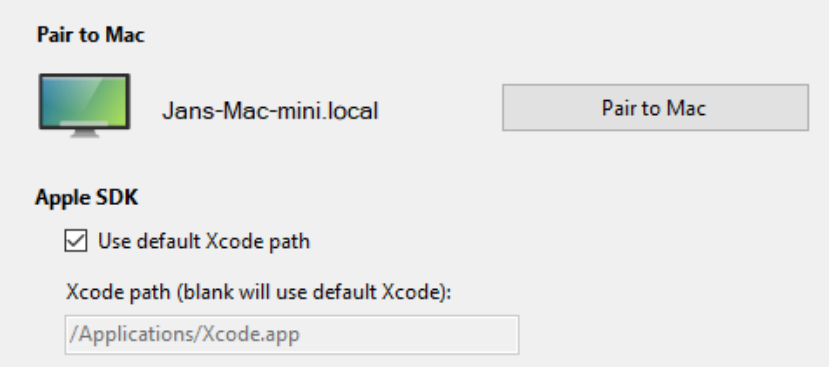
Zaključek namestitve: Na računalniku Windows poženemo Visual Studio. Nato v zgornjem meniju izberemo "Tools", iz spustnega menija izberemo "Options", nato v drevesni strukturi poiščemo razdelek "Xamarin". Znotraj nastavitev za platformo Android preverimo oz. nastavimo poti Android SDK, Android NDK in Java Development Kit, kot je prikazano na sliki 4.5. Znotraj nastavitev za platformo iOS izberemo "Pair to Mac", da vzpostavimo povezavo z računalnikom Mac (slika 4.6). Iz seznama izberemo računalnik Mac, s katerim se želimo povezati, in vnesemo geslo uporabnika.



Slika 4.5: Nastavitve poti za Android.

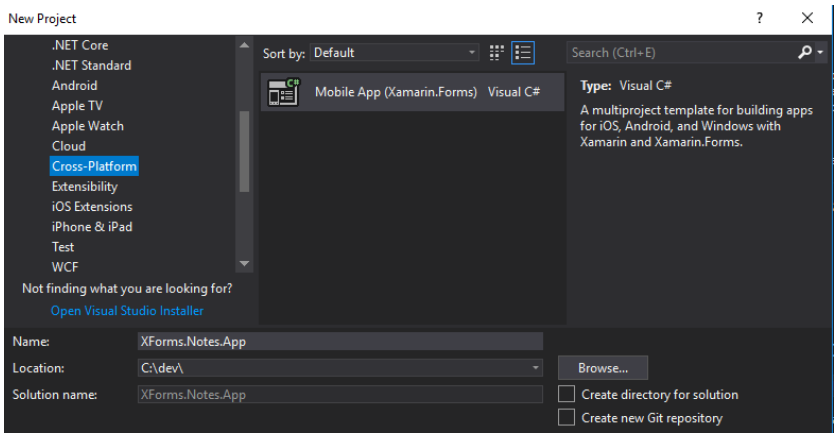
4.3.2 Kreiranje nove programske rešitve

V Visual Studiu v zgornjem meniju izberemo opcijo "File ->New ->Project". Odpre se pogovorno okno, kjer izberemo vrsto projekta in mu nastavimo ime.



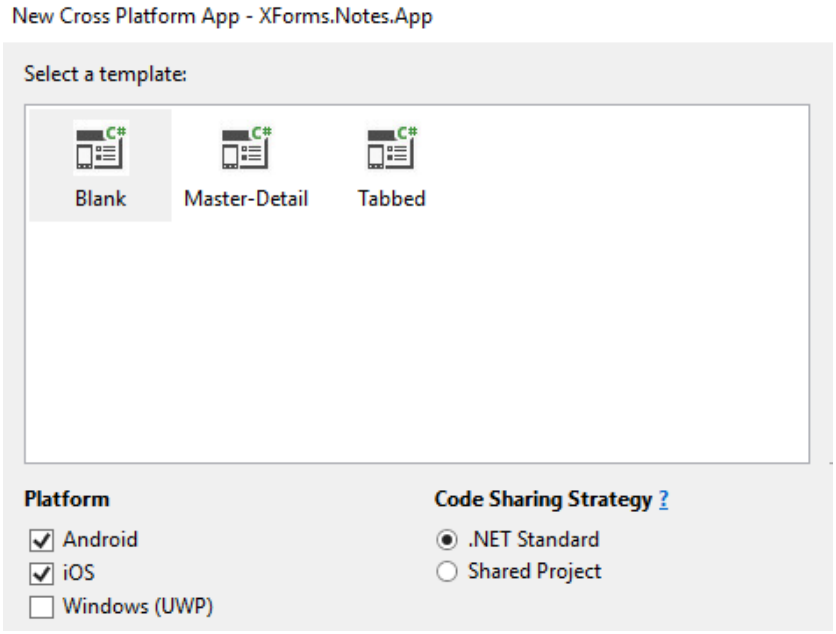
Slika 4.6: Povezava z računalnikom Mac.

Iz drevesne strukture na levi strani izberemo opcijo "Visual C# ->Cross-Platform" in izberemo "Mobile App (Xamarin.Forms)". Spodaj izberemo lokacijo in vnesemo ime projekta oz. rešitve, kot se paketi projektov, ki skupaj predstavljajo končni produkt, v okolju .Net imenujejo (slika 4.7). Po



Slika 4.7: Kreiranje projekta v razvojnem okolju Visual Studio.

potrditvi tega dialoga se odpre nov dialog, kjer izberemo, za katere platforme želimo razvijati in način deljenja kode med platformami, kar prikazuje slika 4.8. Izbiramo lahko med tremi predpripravljenimi predlogami začetnih aplikacij. Izberemo prazno (ang. Blank) predlogo, aplikacije windows (UWP) ne bomo razvijali, zato jo odznačimo in izberemo ".NET Standard" kot strate-



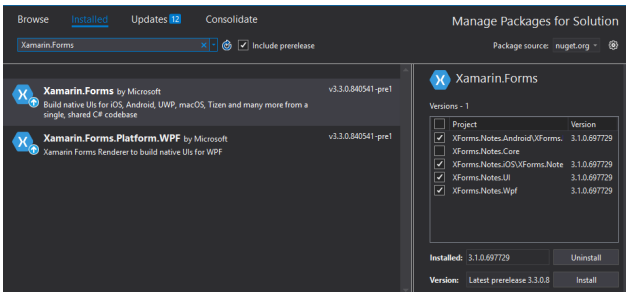
Slika 4.8: Izbira platform in način deljenja kode ob kreiranju projekta Xamarin Forms.

gijo za deljenje kode (ang. Code Sharing Strategy). Dodatne platforme lahko ročno dodamo kasneje. Po potrditvi dialoga bo razvojne okolje ustvarilo novo programsko rešitev s tremi projekti.

Projekt XForms.Notes preimenujemo v XForms.Notes.UI. Za implementacijo poslovne logike aplikacije v programsko rešitev dodamo nov projekt: z desnim klikom na programsko rešitev odpremo kontekstni meni, nato izberemo opcijo "Add->New Project". Za tip projekta izberemo "Class Library (.NET Standard)", poimenujemo ga XForms.Notes.Core in mu nastavimo pravilno lokacijo na disku v mapi, kjer se nahajajo ostali projekti programske rešitve.

Dodamo platformo WPF, ki nam bo pomagala pri hitrejšem razvoju, za kar je potrebnih nekaj korakov. Dodamo nov projekt tipa "WPF App (.NET Framework)" in ga poimenujemo XForms.Notes.Wpf. Odpremo urejevalnik paketov NuGet za programsko rešitev (z desnim klikom na programsko

rešitev odpremo kontekstni meni in izberemo opcijo "Manage NuGet Packages for Solution"), izberemo nameščene pakete, poiščemo Xamarin.Forms paket in ga namestimo na dodan projekt Wpf (slika 4.9). Pomembno je, da

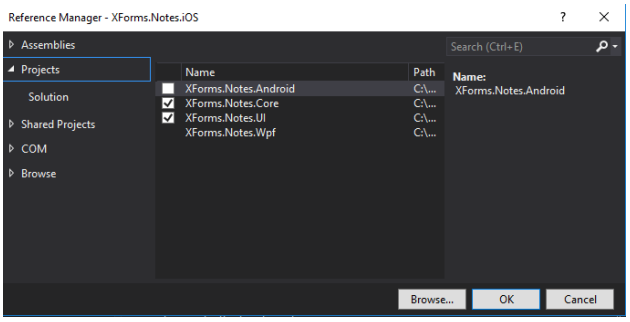


Slika 4.9: Urejevalnik paketov NuGet v razvojnem okolju Visual Studio.

je na vseh projektih nameščena enaka različica Xamarin.Forms paketa.

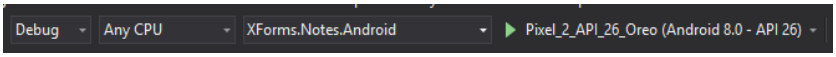
Dodamo še en projekt tipa Shared Project. Poimenujemo ga XForms.Notes.Shared. Namen tega projekta je pojasnjen kasneje.

Projektu Wpf dodamo referenco na projekt UI (desni klik na element "References" znotraj projekta odpre meni, v katerem izberemo opcijo "Add Reference"), kot prikazuje slika 4.10. Projektom Android, iOS, UI in Wpf



Slika 4.10: Nastavljanje referenc projekta v razvojnem okolju Visual Studio.

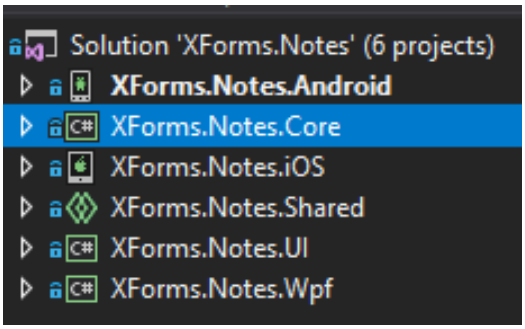
dodamo referenco na projekt Core, projektom Android, iOS in Wpf pa še na projekt Shared. Aplikacijo lahko sedaj poženemo z izbiro zagonskega projekta in naprave ali simulatorja v zgornjem meniju, kar prikazuje slika 4.11.



Slika 4.11: Zagon aplikacije v razvojnem okolju Visual Studio.

4.3.3 Zgradba programske rešitve

Ustvarjena programska rešitev vsebuje šest projektov (slika 4.12), opisanih spodaj.



Slika 4.12: Zgradba programske rešitve.

- **XForms.Notes.Android** je izvršilni projekt aplikacije Android. V tem projektu se nahajajo vstopna točka aplikacije Android in njene osnovne nastavitve. Iz tega projekta lahko dostopamo do Androidovega sistemskega programskega vmesnika in funkcionalnosti ogrodja .NET (Xamarin.Android), ki v .NET Standard knjižnici manjkajo zaradi platformno specifične implementacije.

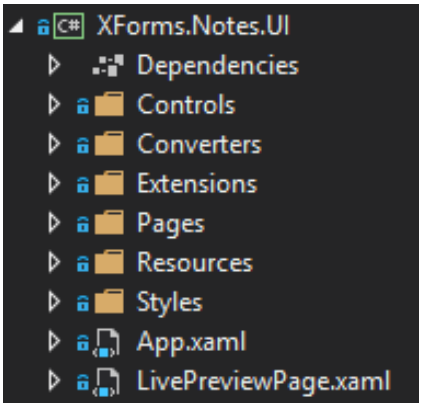
- **XForms.Notes.iOS** je izvršilni projekt aplikacije iOS. V tem projektu se nahajajo vstopna točka aplikacije iOS in njene osnovne nastavitve. Iz tega projekta lahko dostopamo do sistemskega programskega vmesnika iOS in funkcionalnosti ogrodja .NET (Xamarin.iOS), ki v .NET Standard knjižnici manjkajo zaradi platformno specifične implementacije.

- **XForms.Notes.Wpf** je izvršilni projekt Wpf aplikacije Windows. V tem projektu se nahajajo vstopna točka aplikacije Windows in njene osnovne nastavitve. Iz tega projekta lahko dostopamo do Windows sistemskega programskega vmesnika in funkcionalnosti .NET ogrodja (.NET Framework), ki v .NET Standard knjižnici manjkajo zaradi platformno specifične implementacije. Ta projekt sicer ni potreben za razvoj mobilnih aplikacij, uporabljen je samo za hitrejši razvoj, saj se aplikacija Windows Wpf zelo hitro prevede in požene, platforma Wpf pa omogoča tudi spreminjanje kode med samim razhroščevanjem, ne da bi bilo treba aplikacijo ponovno zagnati.

- **XForms.Notes.Shared**. Deljeni projekti v okolju .NET pravzaprav niso pravi projekti, ampak delujejo kot razširitve projekta, ki ima na njih dodano referenco. V njih so dosegljivi vsi imenski prostori, ki so dosegljiv projektu, ki ga razširjajo. Nekatere funkcionalnosti ogrodja .NET so zaradi odvisnosti od platforme v Xamarinu implementirane v platformno specifičnih knjižnicah (Xamarin.iOS in Xamarin.Android). Tak primer so razredi za delo z datotečnim sistemom, ki se nahajajo v imenskem prostoru System.IO. Zaradi uporabe strategije .NET Standard za deljenje kode do njih nimamo dostopa iz skupnih projektov, čeprav je njihova uporaba na vsaki od platform enaka. Za ta namen lahko uporabimo deljen projekt in se s tem znebimo podvajanja kode za vsako platformo.

- **XForms.Notes.UI** je projekt, v katerem je definiran uporabniški vmesnik. Datotečna zgradba projekta je prikazana na sliki 4.13:

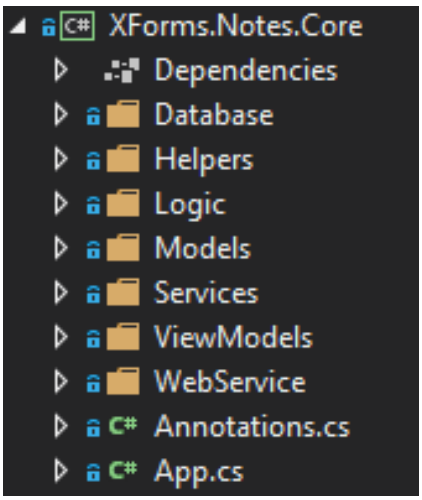
  - mapa Controls vsebuje implementacije lastnih kontrol,
  - mapa Converters vsebuje implementacije pretvornikov vrednosti,
  - mapa Extensions vsebuje implementacije razširitev elementov ogrodja,
  - mapa Pages vsebuje definicije posameznih pogledov aplikacije oz. strani,



Slika 4.13: Datotečna zgradba projekta XForms.Notes.UI.

  - mapa Resources vsebuje uporabljene ikone,
  - mapa Styles vsebuje datoteke, ki definirajo slog aplikacije.

- **XForms.Notes.Core** je projekt, v katerem je implementirana poslovna logika aplikacije. Datotečna zgradba projekta je prikazana na sliki 4.14:



Slika 4.14: Datotečna zgradba projekta XForms.Notes.Core.

- mapa Database vsebuje implementacijo logike za branje in pisanje v podatkovno bazo SQLite,
- mapa Helpers vsebuje implementacije pomožnih metod,
- mapa Logic vsebuje implementacijo logike za upravljanje z uporabniki, zapiski in obvestili,
- mapa Models vsebuje razrede modelov oz. entitet,
- mapa Services vsebuje implementacije storitev,
- mapa ViewModels vsebuje implementacije poglednih modelov,
- mapa WebService vsebuje implementacijo logike za dostop do razvite REST spletne storitve.

4.3.4 MvvmCross

MvvmCross je knjižnica, ki je bila razvita z namenom, da omogoči uporabo arhitekturnega vzorca MVVM razvijalcem, ki razvijajo aplikacije s Xamarinom po klasični metodi. Xamarin Forms prihaja s podporo za razvoj aplikacij po arhitekturnem vzorcu MVVM, knjižnica MvvmCross pa nam olajša delo in doda uporabne funkcionalnosti. Imena razredov knjižnice se začnejo s predpono Mvx. Knjižnica vsebuje za razvoj koristne elemente, nekaj najbolj uporabnih elementov pa je:

- **MvxViewModel** je osnovna implementacija razreda pogledni model z večimi razširitvami. Vsebuje implementacijo vmesnika INotifyPropertyChanged in pomožne metode za nastavljanje in osveževanje lastnosti razreda ter metode pripravljene za lastno implementacijo (ang. Override), ki se izvedejo ob različnih točkah življenskega cikla razreda poglednega modela oz. njegovega pogleda (ang. View).
- **MvxContentPage** je osnovna implementacija strani, ki razširja stran Xamarin.Forms. Glavna prednost tega razreda je, da omogoča nastavitev podatkovnega tipa razreda poglednega modela na generičen način.

- **MvxNavigationService** je implementacija vmesnika IMvxNavigationService in omogoča navigacijo s poglednimi modeli. Navigacija s poglednimi modeli pomeni, da lahko v njih uporabimo instanco navigatorja za navigacijo na naslednji pogledni model. S tem, ko stranem nastavimo podatkovni tip razreda poglednega modela, navigator ve, kateri pogled mora prikazati.

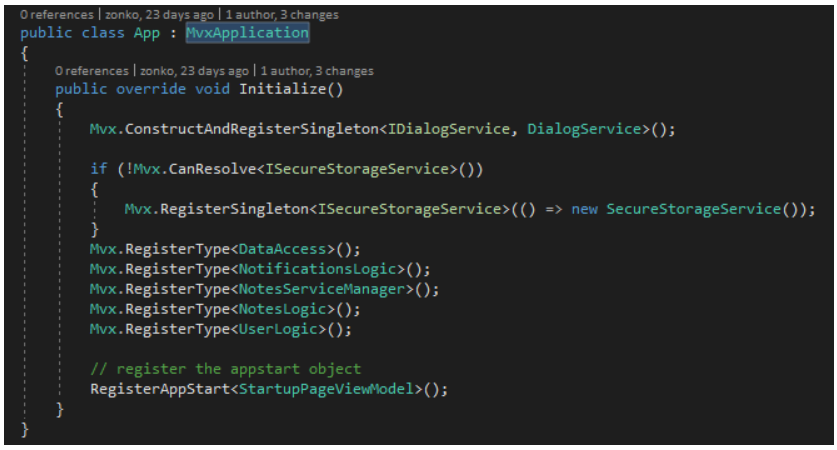
- **Mvx IoC vsebovalnik**. IoC pomeni "Inversion of Control" ali inverzija kontrole. Z vsebovalnikom registriramo podatkovne tipe objektov skupaj z navodili za konstrukcijo in storitve (vmesnike z izbrano implementacijo). Z uporabo vsebovalnika lahko na nivoju platforme registriramo platformno specifično implementacijo vmesnika in do nje, ponovno preko vsebovalnika z uporabo razreševanja, dostopamo v projektih, ki do platformno specifičnih implementacij nimajo neposrednega dostopa.

- **Dependency Injection** ali vbrizgavanje odvisnosti in vsebovalnik za inverzijo kontrole sta tesno povezana. Podatkovne tipe in storitve, ki smo jih registrirali v vsebovalnik, lahko uporabimo kot argumente v konstruktorjih objektov in poglednih modelov. Ob razreševanju registriranih objektov bo vsebovalnik za parametre konstruktorja "vbrizgal" pripravljene instance ali po potrebi skonstruiral nove.

- **MvxMessenger** je implementacija agregatorja dogodkov. Z njim lahko pošiljamo sporočila, ki jih bodo sprejeli vsi objekti, ki so na določen tip sporočila prijavljeni. Na sprejemanje sporočila se lahko v razredih prijavimo na način močne ali šibke reference. Če uporabimo močno referenco, moramo biti pazljivi, da se od sprejemanja sporočil odjavimo, ko naš razred ni več v uporabi.

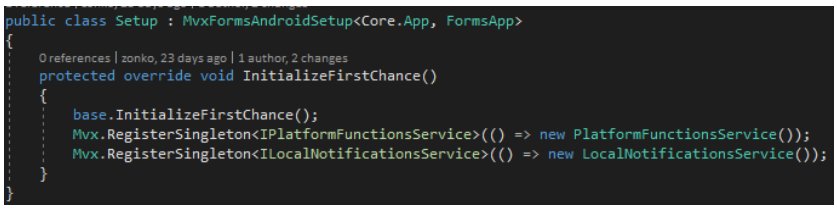
Za uporabo knjižnice v vse projekte, razen XForms.Notes.Core, namestimo paket NuGet, tj. MvvmCross.Forms. V projekt XForms.Notes.Core namestimo paket MvvmCross.Core, v projekt XForms.Notes.Wpf pa dodatno namestimo še MvvmCross.Forms.Platforms.Wpf. Za delovanje je treba v

projekt Core dodati razred App, ki razširja MvxApplication in implementira metodo *Initialize* (slika 4.15). Potrebna je tudi sprememba zagonskih pro-



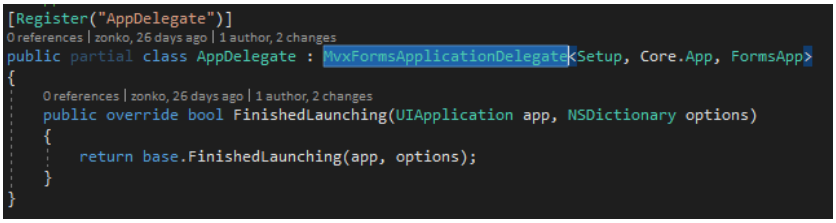
Slika 4.15: Implementacija razreda App v projektu XForms.Notes.Core.

jektov. V vsakega je treba dodati razred Setup v katerem implementiramo metodo *InitializeFirstChance* in v njej v vsebovalnik za inverzijo kotrole registriramo implementacije platformno specifičnih storitev (slika 4.16). Na



Slika 4.16: Implementacija razreda Setup na platformi Android.

koncu popravimo še zagonske razrede na posameznih platformah, kjer podamo podatkovni tip razreda Setup, razreda App, ki smo ga implementirali v projektu Core in razreda FormsApp, ki se nahaja v paketu UI. Na sliki 4.17 je primer za platformo iOS, na ostalih platformah pa je postopek podoben.



Slika 4.17: Implementacija razreda AppDelegate na platformi iOS.

4.3.5 Gradnja uporabniškega vmesnika

Uporabniški vmesnik lahko zgradimo programsko ali pa z uporabo označevalnega jezika XAML. Jezik XAML se povečini uporablja za definicijo uporabniških vmesnikov v .NET okolju za razvoj aplikacij WPF in UWP. Jezik XAML se v Xamarin Forms nekoliko razlikuje od standardnega zaradi drugačne implementacije gradnikov uporabniškega vmesnika.

V Xamarin Forms poznamo štiri osnovne gradnike:

- **Page** ali stran. Predstavlja en zaslon aplikacije, podobno kot Activity v razvoju za android. Obstaja več vrst strani:
  - ContentPage – stran z vsebino,
  - MasterDetailPage – stran z navigacijskim predalom,
  - NavigationPage – stran z navigacijsko vrstico,
  - TabbedPage – stran z zavihki,
  - CarouselPage – stran z drsnim premikanjem (vrtiljak).
- **View** ali pogled. Skoraj vsi ostali gradniki (gumbi, vnosna polja ...) izhajajo iz pogleda.
- **Layout** ali razvrščevalnik. Razvrščevalniki so posebni pogledi, ki se uporabljajo za razvrščanje elementov na zaslon. Primeri razvrščevalnikov so:

  - StackLayout – sklada elemente horizontalno ali vertikalno,
  - AbsoluteLayout – razvršča elemente glede na absolutno določeno pozicijo,
  - RelativeLayout – razvršča elemente glede na relativno določeno pozicijo,
  - Grid – razvršča elemente v mrežo definirano z določitvijo stolpcev in vrstic, omogoča skladanje elementov v smeri Z-osi; v primeru prekrivanja elementov je viden zadnji dodan element,
  - FlexLayout – sklada elemente horizontalno ali vertikalno s prelivanjem v naslednjo vrstico ali stolpec, ko zmanjka prostora,
  - ScrollView – drsni pogled.

- **ViewCell** ali celica pogleda. Predstavlja celico v seznamu ali tabeli.

Primer za dodajanje strani za prijavo

Najprej dodamo razred LoginPageViewModel v mapo ViewModels v projektu XForms.Notes.Core. Nastavimo mu osnovni razred MvxViewModel in mu dodamo potrebne lastnosti, ki jih uporabljamo v definiciji pogleda z uporabo podatkovne vezave (slika 4.18). Lastnosti tipa Command uporabimo za izvrševanje akcije, ko uporabnik klikne na gumb. V primeru *LoginCommand* iz slike 4.18 je *Login* privatna asinhrona metoda tipa Task, ki izvede potrebno logiko za prijavo uporabnika.

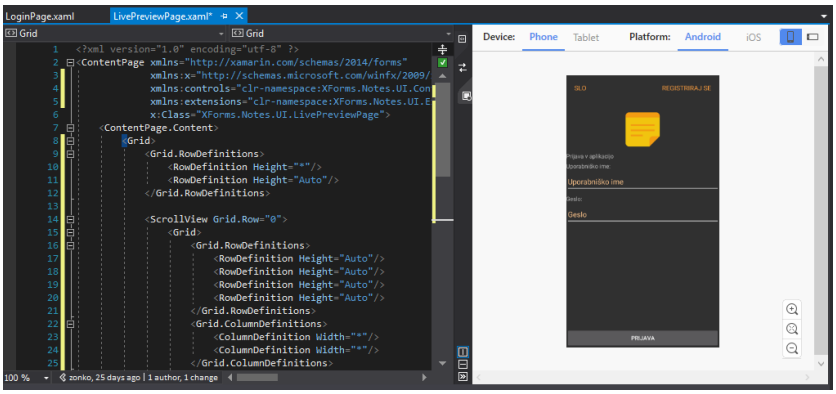
```
2 references | zonko, 28 days ago | 1 author, 1 change
public string Password
{
    get => _password;
    set => SetProperty(ref _password, value);
}

0 references | zonko, 26 days ago | 1 author, 2 changes
public MvxAsyncCommand LoginCommand => _loginCommand ?? (_loginCommand = new MvxAsyncCommand(Login));
```

Slika 4.18: Primer lastnosti poglednega modela.

Stran dodamo v aplikacijo z desnim klikom na mapo "Pages" v projektu

XForms.Notes.UI, nato iz menija izberemo "Add" in nato "NewItem". Izberemo "ContentPage" in vnesemo ime – LoginPage. Zaradi navigacije z uporabo poglednih modelov je dodani strani treba spremeniti osnovni razred iz ContentPage v MvxContentPage. To storimo tako, da v datoteki .xaml zamenjamo ContentPage z MvxContentPage. Pri tem zelo pomaga uporaba razširitve za Visual Studio: ReSharper. Brez uporabe ReSharperja je treba še ročno dodati referenco na imenski prostor, kjer se MvxContentPage nahaja.

Za predogled izgleda strani, ne da bi pognali aplikacijo, lahko uporabimo LivePreview v urejevalniku datotek .xaml. Predogled ne deluje, če strani zamenjamo osnovni razred, zato lahko v projekt dodamo stran, namenjeno razvoju z uporabo živega predogleda, ki ji ne spremenimo osnovnega pogleda (slika 4.19). Primer kode XAML dokončane strani za registracijo je prikazan

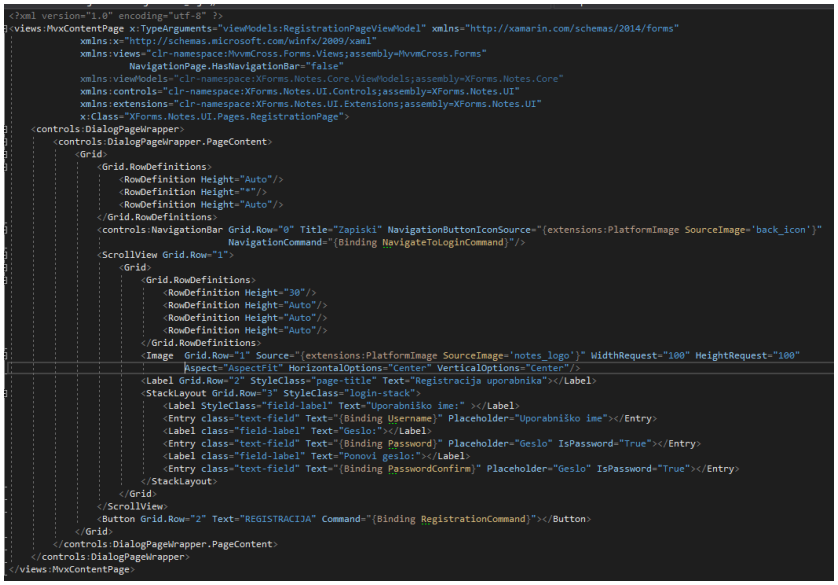


Slika 4.19: Živi predogled strani.

na sliki 4.20.

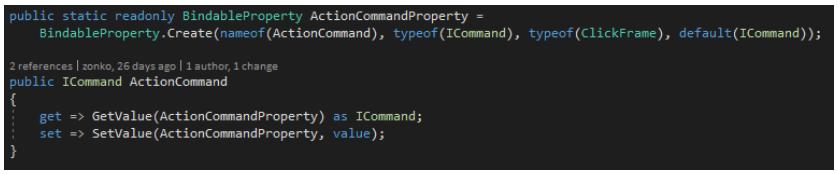
Lastne in po meri zgrajene kontrole

Če v uporabniškem vmesniku pogosto uporabljamo enak skupek gradnikov, jih lahko ločimo v ločeno datoteko .xaml in na ta način zgradimo lastno kontrolo (ang. User Control). Če pa potrebujemo gradnik, ki v ogrodju ni prisoten, pa lahko poskusimo gradnik zgraditi kot po meri zgrajeno kontrolo. Če v gradniku potrebujemo še dodatne funkcionalnosti, ki jih je treba im-



Slika 4.20: Koda XAML dokončane strani za registracijo.

plementirati na posamezni platformi, je potrebna implementacija lastnega izrisovalnika (ang. Custom Renderer) oziroma njegove razširitve.



Slika 4.21: Primer lastnosti, ki omogoča podatkovno vezavo.

NavigationBar: Primer lastne kontrole v naši aplikaciji je navigacijska vrstica. Zgrajena lastna kontrola NavigationBar je sestavljena iz dveh gumbov na vsaki strani in besedila za naslov na sredini. Ob uporabi v jeziku XAML omogoča podatkovno vezavo akcij obeh gumbov in nastavitev naslova ter ikon gumbov (slika 4.21). Kontrola je uporabljena na skoraj vseh straneh aplikacije.

DialogPageWrapper: Xamarin Forms nima vgrajenega načina za prikazovanje dialogov. Lahko bi uporabili eno izmed knjižnic, vendar večinoma

ne omogočajo veliko možnosti nastavitev sloga dialoga. Zato zgradimo lastno kontrolo, ki bo uporabljena kot osnovni element na vseh straneh. Kontrola je sestavljena z uporabo mrežnega razvrščevalnika z eno celico, v kateri se nahajajo: vsebina strani, ki jo je mogoče nastaviti preko lastnosti PageContent, indikator obdelave in vsebina dialoga z gumbom. Za delovanje kontrola uporablja podatkovno vezavo na za ta namen razvit DialogService, ki je ob zagonu registriran v vsebovalnik za inverzijo kontrole.

StackList: Xamarin Forms omogoča uporabo gnezdenih drsnih pogledov. Primer je uporaba drsnega pogleda za prikaz celotne vsebine strani in uporaba seznama (ListView) znotraj te strani. Seznam vsebuje lasten drsni pogled. Tak primer je v naši aplikaciji na strani za urejanje zapiska, kjer je prikazan seznam priponk. Problem pa je, da se ob malo večjem številu elementov seznam začne čudno obnašati (nastane prazen prostor pod njim ali pa ne omogoča pregleda vseh elementov). Zato razvijemo po meri zgrajeno kontrolo, ki bo omogočala podatkovno vezavo seznama elementov in jih prikazala s skladanjem, brez drsnega pogleda, zunanji drsni pogled pa bo omogočal pregled vseh elementov. Pomemben element kontrole je lastnost ItemTemplate tipa DataTemplate, ki jo je mogoče nastaviti v kodi XAML in predstavlja predlogo celice seznama (slika 4.22).

ClickFrame: Problem v Xamarin Forms je, da lahko akcije uporabnika na dotik elementa uporabljamo samo na izbranih elementih, na primer gumbih. ClickFrame je implementacija okvirja z dodano funkcionalnostjo. Za osnovni razred uporablja Frame in ima dodatno lastnost Command, ki omogoča podatkovno vezavo in dogodek Clicked za uporabo brez podatkovne vezave. Oba se izvedeta, ko je bil zaznan klik. Za delovanje je potrebna implementacija razširitve izrisovalnika za element Frame na vseh podprtih platformah. Izrisovalnik služi kot povezava med Xamarin Forms gradniki in domorodnimi gradniki. Vsak gradnik v Xamarin Forms ima implementacijo svojega izrisovalnika na vseh platformah. V izrisovalnikih je definirano, kateri domorodni element naj se uporabi za prikaz določenega elementa Xamarin Forms. Kljub imenu pa izrisovalnik skrbi tudi za komunikacijo med domo-

```
5 references | zonko, 24 days ago | 1 author, 1 change
public class StackList : StackLayout
{
    public static readonly BindableProperty ItemsSourceProperty =
        BindableProperty.Create("ItemsSource", typeof(IEnumerable), typeof(StackList), default(IEnumerable), prop

    1 reference | zonko, 24 days ago | 1 author, 1 change
    public IEnumerable ItemsSource
    {
        get => (IEnumerable)GetValue(ItemsSourceProperty);
        set => SetValue(ItemsSourceProperty, value);
    }

    public static readonly BindableProperty ItemTemplateProperty =
        BindableProperty.Create("ItemTemplate", typeof(DataTemplate), typeof(StackList), default(DataTemplate));

    2 references | zonko, 24 days ago | 1 author, 1 change
    public DataTemplate ItemTemplate
    {
        get => (DataTemplate)GetValue(ItemTemplateProperty);
        set => SetValue(ItemTemplateProperty, value);
    }

    1 reference | zonko, 24 days ago | 1 author, 1 change
    private static void OnItemsSourceChanged(BindableObject bindable, object oldvalue, object newvalue)
    {
        if(!(bindable is StackList stackList) || stackList.ItemTemplate == null) return;

        stackList.Children.Clear();

        foreach (var item in stackList.ItemsSource)
        {
            var viewCell = stackList.ItemTemplate.CreateContent() as ViewCell;
            if (viewCell == null) continue;
            viewCell.View.BindingContext = item;
            stackList.Children.Add(viewCell.View);
        }
    }
}
```

Slika 4.22: Implementacija kontrole StackList.

rodnim gradnikom in gradnikom Xamarin Forms. Tako lahko v razširjenem izrisovalniku poslušamo na dogodek klik na domorodnem elementu in ga posredujemo naši kontroli (slika 4.23).

Slog aplikacije

Za nastavljanje sloga aplikacije lahko na posameznih elementih v kodi XAML nastavimo lastnosti, kot so barva ozadja, barva besedila in drugo. Boljši način pa je, da definiramo globalne sloge. To lahko naredimo na dva načina: prvi način je, da v datoteko App.xaml dodamo slovar virov (ang. ResourceDictionary). V ta slovar vnesemo definicije slogov XAML, ki jim določimo lastnosti in vrsto elementa, ki mu je slog namenjen. V slovar lahko dodamo tudi statične vire, kot na primer barve in pretvornike vrednosti, ki jih lahko potem uporabimo v celotni aplikaciji. Z uporabo značke OnPlatform lahko določimo tudi vrednosti, specifične za posamezno platformo.

Drug način je uporaba datotek CSS. Zadnja različica Xamarin Forms nam

```
[assembly: ExportRenderer(typeof(ClickFrame), typeof(ClickFrameRenderer))]
namespace XForms.Notes.Droid.CustomRenderers
{
    2 references | zonko, 24 days ago | 1 author, 2 changes
    public class ClickFrameRenderer : FrameRenderer
    {
        0 references | zonko, 26 days ago | 1 author, 1 change
        public ClickFrameRenderer(Context context) : base(context)
        {
        }

        0 references | zonko, 24 days ago | 1 author, 2 changes
        protected override void OnElementChanged(ElementChangedEventArgs<Frame> e)
        {
            base.OnElementChanged(e);
            var newElement = e.NewElement as ClickFrame;
            var oldElement = e.OldElement as ClickFrame;
            if (newElement != null && oldElement == null)
            {
                Click += newElement.OnClick;
            }
            if (newElement == null && oldElement != null)
            {
                Click -= oldElement.OnClick;
            }
        }
    }
}
```

Slika 4.23: Implementacija razširitve izrisovalnika na platformi Android.

omogoča uporabo omejenega nabora lastnosti CSS. Z uporabo znaka (^) pred selektorjem lahko povemo, da želimo, da se slog uporabi na vseh elementih, katerih ime osnovnega razreda je podano za tem znakom. Datoteko CSS vključimo v slovar virov.

Zagonski zaslon in ikone aplikacije določimo posebej na vsako platformo. Postopek je podoben kot pri razvoju platformno specifičnih aplikacij.

Za Android v mapo Resources v projektu XForms.Notes.Android dodamo ikone v podmape minimap, v mapo layout pa dodamo datoteko XML v kateri definiramo zagonski zaslon. V datoteki styles.xml v mapi values nastavimo temo zagonskega zaslona in aplikacije. Nastaviti je treba tudi osnovne barve v datoteki colors.xml.

Za iOS odpremo datoteko Assets, ki se nahaja v paketu Asset Catalogs v XForms.Notes.iOS projektu, in s pomočjo grafičnega urejevalnika nastavimo zagonske slike in ikone. Popravimo še datoteko LaunchScreen.storyboard v mapi Resources.

Uporaba ikon in slik: Najboljši način je, da ikone in slike z enakim imenom skopiramo v mapo Resources v projektu iOS in v mape drawable v projektu Android. Na ta način lahko določimo različne velikosti slik za različne zaslone. Prikažemo jih lahko z uporabo elementa Image, ki mu za lastnost Source nastavimo ime slike brez končnice.

Drug način je uporaba vgrajenih virov v projektu UI, za ta namen je potrebna implementacija razširitve (slika 4.24).

```
[Preserve(AllMembers = true)]
[ContentProperty(nameof(Source))]
1 reference | zonko, 26 days ago | 1 author, 1 change
public class ImageResourceExtension : IMarkupExtension
{
    3 references | zonko, 26 days ago | 1 author, 1 change
    public string Source { get; set; }
    1 reference | zonko, 26 days ago | 1 author, 1 change
    public object ProvideValue(IServiceProvider serviceProvider)
    {
        if (Source == null) return null;
        var imageSource = ImageSource.FromResource(Source, typeof(ImageResourceExtension).GetTypeInfo().Assembly);
        return imageSource;
    }
}

Source="{extensions:ImageResource Source='notes_logo'}"
```

Slika 4.24: Implementacija in uporaba značkovne razširitve za prikaz slik z uporabo vgrajenih virov.

Komunikacija med zasloni

Za komunikacijo med zasloni lahko uporabimo vgrajene funkcionalnosti knjižnice MvvmCross.

Če želimo, da zaslon ob inicializaciji sprejme argument, lahko za njegov pogledni model uporabimo generični razred MvxViewModel z določenim podatkovnim tipom argumenta. Tipi argumentov so omejni na preproste tipe, saj se v ozadju na platformi Android uporabi namera (ang. Intent) z nastavljenimi dodatnimi argumenti (ang. Extras). Za tako ustvarjen pogledni model moramo implementirati prepis metode *Prepare*. Argument podamo ob navigaciji na nov zaslon. V naši aplikaciji je tak primer zaslon za urejanje zapiska, ki sprejme ID zapiska.

Za zaslone, od katerih pričakujemo odgovor, za njihov pogledni model uporabimo generični osnovni razred MvxViewModelResult, ki mu podamo

podatkovni tip odgovora. Take zaslone zapremo s klicem metode *Close* na navigatorju, zaslon, ki je sprožil navigacijo na tak zaslon, pa bo v tem trenutku dobil odgovor z rezultatom, ki je bil podan metodi *Close*. Tak primer v naši aplikaciji je zaslon za nastavljanje datuma in ure obvestila, ki izbran čas vrne zaslonu za urejanje zapiska.

Za komunikacijo med dvema aktivnima zaslonoma ali med posameznimi elementi enega zaslona laho uporabimo MvxMessenger.

4.3.6 Delo s podatkovno bazo SQLite

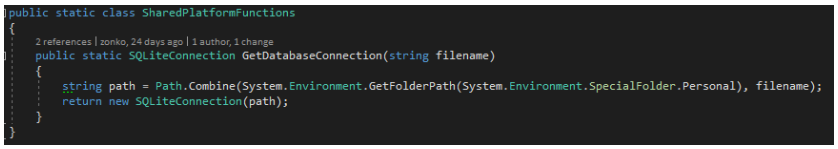
V naši aplikaciji podatke shranjujemo v podatkovno bazo SQLite. SQLite je relacijska podakovna baza, ki za delovanje ne potrebuje strežnika, zato je zelo primerna za uporabo v mobilnih aplikacijah.

Podpora za uporabo podatkovne baze SQLite ni vgrajena v Xamarin Forms. Potrebna je uporaba knjižnice.

Na voljo je več različnih knjižnic, najboljša med njimi pa je SQLite-Net Extensions, ki za delovanje uporablja knjižnico SQLite-net-pcl. Knjižnica za delo s podatkovno bazo uporablja metodo ORM (ang. Object Relational Mapping), kar pomeni, da lahko s podatkovno bazo upravljamo z uporabo objektov jezika C#. Za poizvedovanje lahko uporabimo LINQ (ang. Language Integrated Query), ki je vgrajen v jezik C# in omogoča poizvedovanje po vseh vrstah podprtih seznamov. Poleg tega knjižnica po metodi ORM omogoča tudi kaskadne operacije, kot sta zajem objekta z vsemi podatki iz odvisnih tabel in kaskadno posodabljanje ter brisanje zapisa. Zaradi uporabe LINQ in metode ORM je možnost napak veliko manjša, saj se znebimo vseh tipkarskih napak ob pisanju poizvedb SQL.

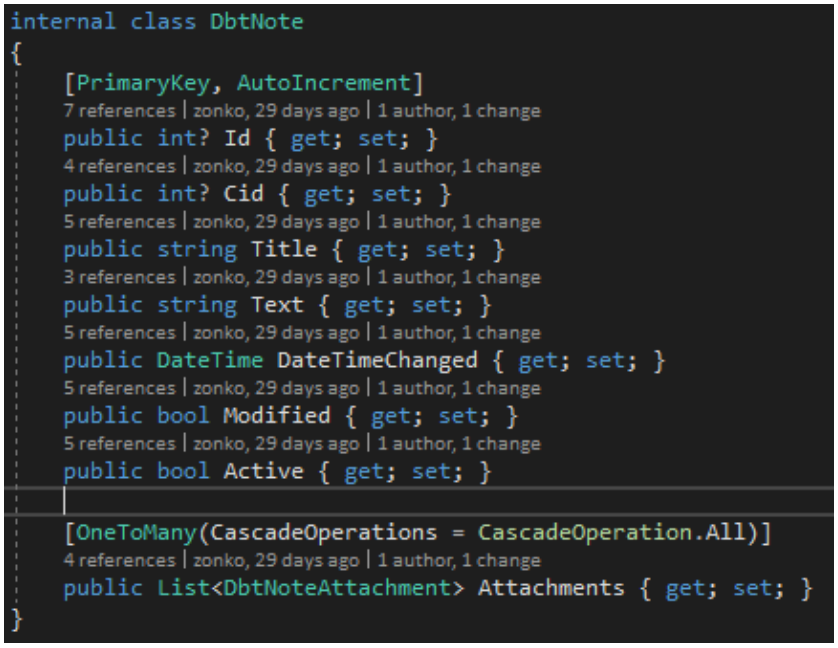
Za uporabo knjižnice najprej namestimo paket NuGet SQLite-Net Extensions v projekte Android, iOS, Wpf in Core. Odpiranje povezave na podatkovno bazo je platformno specifična operacija, saj potrebuje dostop do datotečnega sistema platforme. Za ta namen v projekt Core dodamo vmesnik IPlatformFunctionsService, ki vsebuje metodo *GetDatabaseConnection*. Vmesnik implementiramo na vsaki platformi ter ga registriramo v vseboval-

nik IoC v razredu Setup. Ker je naša implementacija odpiranja povezave na podatkovno bazo na vseh platformah enaka, jo lahko prestavimo v deljen projekt in se na njo sklicujemo v platformno specifični implementaciji (slika 4.25). Odpiranje povezave bo podatkovno bazo ustvarilo, če ta ne ob-



Slika 4.25: Implementacija odpiranja povezave v deljenem projektu.

staja. Tabele definiramo z razredi, katerih lastnosti opremimo z atributi, ki določajo primarne ključe, relacije, tuje ključe in drugo. (slika 4.26)



Slika 4.26: Definicija tabele DbNote z uporabo razreda.

V naši aplikaciji smo za delo s podatkovno bazo implementirali razred DataAccess. Razred vsebuje metodi za inicializacijo uporabnikove in aplikacijske podatkovne baze. Uporabnikova podatkovna baza se inicializira

z uporabniškim imenom in vsebuje tabele za shranjevanje zapiskov, priponk zapiskov in tabelo za shranjevanje nastavitev po načinu ključ-vrednost (slika 4.27).

```
internal async Task InitializeUserDatabase()
{
    using (var connection = _platformFunctionsService.GetDatabaseConnection(await GetUserDatabaseFilename()))
    {
        connection.CreateTable<DbtKeyValue>();
        connection.CreateTable<DbtNote>();
        connection.CreateTable<DbtNoteAttachment>();
    }
}
```

Slika 4.27: Inicializacija uporabniške podatkovne baze (metoda *CreateTable* posodobi model tabele oz. jo ustvari, če ne obstaja).

Aplikacijska podatkovna baza vsebuje samo tabelo, v kateri so shranjena obvestila. Ostale metode so namenjene pridobivanju in shranjevanju podatkov. Na sliki 4.28 je prikazana implementacija metode za pridobivanje seznama zapiskov, ki jih je treba sinhronizirati s strežnikom.

```
1 reference | zonko, 24 days ago | 1 author, 3 changes
internal async Task<List<Note>> ListModifiedNotes()
{
    using (var connection = _platformFunctionsService.GetDatabaseConnection(await GetUserDatabaseFilename()))
    {
        return connection.GetAllWithChildren<DbtNote>(n => n.Modified, true)
            .Select(n => new Note
            {
                Id = n.Id,
                DateTimeChanged = n.DateTimeChanged,
                Title = n.Title,
                Active = n.Active,
                Cid = n.Cid,
                Modified = n.Modified,
                Text = n.Text,
                Attachments = n.Attachments?.Select(att => new NoteAttachment
                {
                    Content = att.Content,
                    Name = att.Name,
                    Type = att.Type
                }).ToList()
            }).ToList();
    }
}
```

Slika 4.28: Implementacija metode za pridobivanje podaktov iz podatkovne baze SQLite.

4.3.7 Uporaba spletne storitve REST

Razvita spletna storitev REST za prenos podatkov uporablja strukturo JSON (ang. JavaScript Object Notation). Za uspešno komunikacijo mora naša aplikacija znati uporabljati te podatke. To najlažje dosežemo z uporabo serializacije JSON. Za uporabo serializacije JSON pa potrebujemo knjižnico Newtonsoft.Json, ki pa nam je ni treba posebej nameščati, saj se je namestila skupaj s paketom SQLite-Net Extensions, ki jo tudi uporablja.

Najprej ustvarimo razrede, ki so reprezentacije objektov JSON, ki se bodo prenašali z uporabo spletne storitve. Njihove lastnosti opremimo z atributi JsonProperty, ki povedo, kakšna so imena posameznih objektov v JSON strukturi (slika 4.29). To je sicer potrebno samo zato, ker konvencija poimenovanja lastnosti v C# določa, da se te začnejo z veliko začetnico, storitev pa nam vrača imena objektov z malo začetnico.

V naši aplikaciji imamo vse metode, namenjene uporabi spletne storitve, implementirane v razredu NotesServiceManager. V tem razredu je določen tudi spletni naslov strežnika in imena metod spletne storitve. Poleg metod za prijavo, registracijo in sinhronizacijo je tukaj še metoda *Ping* in privatna metoda *PostGet*.

Metoda *Ping* je namenjena preverjanju dosegljivosti strežnika in se izvede pred vsakim klicem ostalih metod. Ob njenem klicu se nastavi časovna omejitev na tri sekunde (privzeta nastavitev je 120 sekund). Če strežnik ni dosegljiv, nam tako ni treba čakati dve minuti, preden nadaljujemo z izvajanjem aplikacije.

Metoda *PostGet* pa je zasebna in je namenjena dejanski vzpostavitvi povezave in prenosu podatkov. Za delovanje uporablja WebRequest, ki mu pred klicem strežnika nastavi zaglavne podatke, kot so ContentType in Authorization, kamor se nastavi avtentikacijski žeton uporabnika, pridobljen ob prijavi.

```
internal class WsNote
{
    [JsonProperty("id")]
    2 references | zonko, 29 days ago | 1 author, 1 change
    public int? Id { get; set; }

    [JsonProperty("clientId")]
    2 references | zonko, 29 days ago | 1 author, 1 change
    public int? ClientId { get; set; }

    [JsonProperty("active")]
    2 references | zonko, 29 days ago | 1 author, 1 change
    public bool Active { get; set; }

    [JsonProperty("title")]
    2 references | zonko, 29 days ago | 1 author, 1 change
    public string Title { get; set; }

    [JsonProperty("text")]
    2 references | zonko, 29 days ago | 1 author, 1 change
    public string Text { get; set; }

    [JsonProperty("dateTimeChanged")]
    2 references | zonko, 29 days ago | 1 author, 1 change
    public string DateTimeChanged { get; set; }

    [JsonProperty("attachments")]
    2 references | zonko, 29 days ago | 1 author, 1 change
    public List<WsNoteAttachment> Attachments { get; set; }
}
```

Slika 4.29: Definicija razreda za serializacijo v obliko JSON.

4.3.8 Varno shranjevanje podatkov

Podatki, ki so shranjeni v podatkovni bazi SQLite niso popolnoma varno shranjeni. Dostop do datoteke podatkovne baze je sicer težaven, vendar ni nemogoč, zato podatkovna baza ni primerna za shranjevanje občutljivih podatkov, kot so gesla. V naši aplikaciji sicer gesel uporabnikov ne shranjujemo, shranjujemo pa avtentikacijski žeton uporabnika za avtentikacijo s spletnim strežnikom. Žeton se ob uspešni prijavi uporabnika shrani z namenom, da se mu naslednjič ob odpiranju aplikacije ni treba ponovno prijavljati.

Za varno shranjevanje podatkov uporabimo knjižnico Secure Storage iz

paketa Xamarin.Essentials. Namestimo jo v projekte Android, iOS in Core z uporabo urejevalnika paketov NuGet.

Knjižnica za varno shranjevanje podatkov na platformi iOS uporablja iOS KeyChain, zato je treba v projektu iOS v datoteki Entitlements.plist omogočiti uporabo.

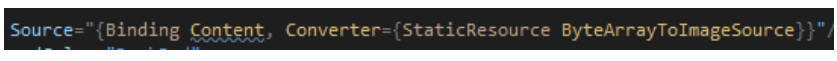
Na Platformi Android knjižnica uporablja Android KeyStore za shranjevanje ključa, s katerim zakriptira podatke in jih nato shrani v deljeno shrambo (ang. Shared Preferences).

4.3.9 Dodajanje slik

V aplikaciji lahko na zapisek dodamo priponko oz. sliko. Za dodajanje slike jo moramo najprej izbrati na napravi, za kar je bil v naši aplikaciji uporabljen vtičnik MediaPlugin. Vtičnik namestimo v projekte iOS, Android in Core z uporabo urejevalnika paketov NuGet. Pred uporabo vtičnika je potrebnih nekaj dodatnih prilagoditev za vsako platformo. Natančnejši opis potebnih sprememb se nahaja na github strani vtičnika [9].

Uporaba vtičnika je enostavna. Vse metode vtičnika so asinhrone. Do njih dostopamo preko instance razreda, ki se nahaja v razredu CrossMedia.Current. Pred izbiro slike moramo pognati inicializacijo s klicem metode *Initialize*. Metoda *PickPhotoAsync* odpre izbirnik slik na platformi in vrne objekt tipa MediaFile. Podamo ji lahko tudi argument, ki določa maksimalno velikost slike in njeno stiskanje. Objekt tipa MediaFile lahko preberemo v podatkovni tok in ga nato pretvorimo v polje zlogov.

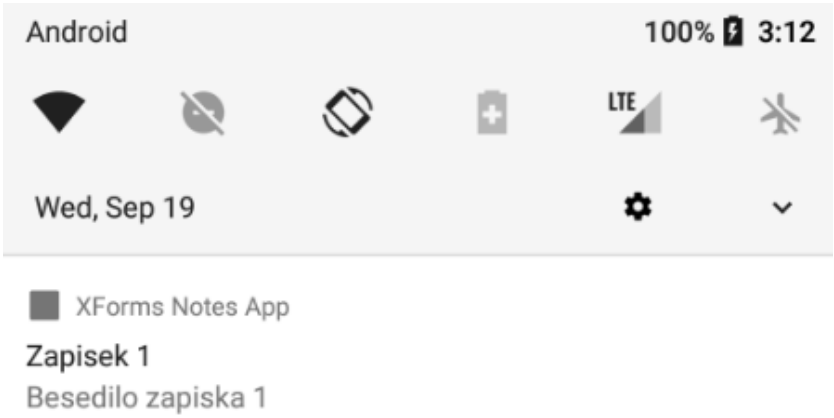
Polje zlogov shranimo v podatkovno bazo, za prikaz na zaslon pa uporabimo element Image s podatkovno vezavo lastnosti Source in pretvornikom vrednosti (slika 4.30).



Slika 4.30: Uporaba pretvornika za prikaz slike z uporabo polja zlogov.

4.3.10 Lokalna obvestila

Za prikaz lokalnih obvestil v Xamarin Forms obstaja kar nekaj knjižnic. Nobena pa ne omogoča nastavitve odzivne metode, ki bi se izvedla, ko je uporabnik kliknil na obvestilo. V naši aplikaciji hočemo ob kliku obvestila odpreti zaslon za urejanje zapiska, ki ga obvestilo zadeva. Zato je bila za ta namen implementirana lastna rešitev na podlagi navodil za lokalna obvestila na straneh dokumentacije Xamarina. Implementacija je specifična za vsako platformo posebej, na obeh pa implementira vmesnik in se ob zagonu aplikacije registrira kot storitev v vsebovalnik za inverzijo kontrole. Najpomembnejša elementa sta metoda *Show* in dogodek NotificationOpened. Metoda *Show* kot argumente sprejme naslov, besedilo, čas obestila in ID obvestila, po katerem vemo, s katerim uporabnikom in zapiskom je obvestilo povezano (slika 4.31). Dogodek NotificationOpened se sproži, ko je bilo obvestilo od-



Slika 4.31: Obvestilo na platformi Android.

prto, nanj je prijavljena metoda v razredu NotificationsLogic, ki po potrebi izvede ustrezno akcijo.

4.3.11 Lokalizacija

Za lokalizacijo aplikacije lahko uporabimo datoteke virov (ang. Resource File) oz. datoteke .resx. Datoteke virov dodamo v projekt Core. Najprej dodamo datoteko namenjeno privzetemu jeziku in jo poimenujemo Language. Naslednjim datotekam dodamo na konec imena še oznako jezika, ki ga predstavlja. Oznaka za slovenski jezik je "sl-SI". Datoteke napolnimo s ključi in vrednostmi s pomočjo grafičnega urejevalnika. Ključi morajo biti v vseh datotekah enaki.

V kodi C# lahko do vrednosti dostopamo s sklicevanjem na ključ v objektu Language: Language.Kljuc. V uporabniškem vmesniku lahko uporabimo podatkovno vezavo z indeksom, za kar moramo v poglednih modelih dodati lastnost, ki takšno vezavo omogoča.

Za menjavo jezika objektu Language nastavimo želeno kulturo z uporabo metode *SetCulture*. Za osvežitev uporabniškega vmesnika mu je treba sporočiti, da je prišlo do spremembe na lastnosti poglednega modela, to storimo z uporabo metode *RaisePropertyChanged*.

Poglavje 5

NativeScript

Ogrodje NativeScript je razvilo podjetke Telerik, ki se je večinoma ukvarjalo z razvojem komponent uporabniških vmesnikov za spletne, mobilne in aplikacije .NET, in je od leta 2014 del podjetja Progress. Prva različica je izšla maja leta 2014 pod licenco Apache 2.0. NativeScrtip je s tem postal prostodostopno ogrodje za hkraten navzkrižni razvoj mobilnih aplikacij za platformi iOS in Android. Prva različica je omogočala razvoj aplikacij v programskem jeziku JavaScript in gradnjo uporabniškega vmesnika v označevalnem jeziku XML.

Leta 2016 je izšla različica 2.0 katere glavne novosti so bile integracija z aplikacijskim ogrodjem Angular, živo osveževanje pognane aplikacije (ang. Hot reload) in podpora programskega jezika TypeScript. TypeScript je nadnabor (ang. Superset) programskega jezika JavaScript, omogoča uporabo razredov, modulov, generikov, vmesnikov, enumeratorjev... in se ob uporabi prevaja v JavaScript (ang. Transpile). Razvil ga je Microsoft z namenom, da bi omilil slabosti in olajšal uporabo programskega jezika JavaScript.

Poleg ogrodja Angular pa NativeScript omogoča tudi uporabo ogrodja Vue.js za katerega vtičnik je razvila skupnost.

5.1 Način delovanja

NativeScript za izvajanje aplikacij uporablja izvajalni okolji JavaScript. Na platformi Android je to Googlovo izvajalno okolje V8 JavaScript Virtual Machine, ki ga za izvajanje JavaScript kode uporablja tudi spletni brskalnik Chrome. Na platformi iOS za izvajanje aplikacij NativeScript skrbi v iOS vgrajeno izvajalno okolje JavaScriptCore. Prevedena oz. spakirana aplikacija se izvaja neposredno v teh izvajalnih okoljih brez dodatnega prevajanja v kodo, specifično za posamezno platformo. Ob pakiranju aplikacije pa se glede na ciljno platformo zamenjajo le elementi grafičnega vmesnika. Ob pakiranju aplikacije za Android se tako za gumb, določen v definiciji XML uporabniškega vmesnika, vzame com.android.widget.Button, ob pakiranju za iOS pa UIButton.

Za dostop do sistemskega programskega vmesnika kot tudi domorodnih knjižnic NativeScript uporablja odsevnik (ang. Reflection), s katerim ob zagonu aplikacije pridobi informacije o dosegljivih metodah. Na ta način se lahko v skupni kodi sklicujemo na metode programskega vmesnika platforme, NativeScript pa bo na podlagi imena metode ali objekta in njenih argumentov ugotovil, da mora za izvajanje uporabiti klic metode sistemskega programskega vmesnika. Pri tem moramo biti pazljivi, da kodo, specifično za platformo, ločimo npr. z uporabo if stavka: *if (page.ios)* { }. Zavedati pa se moramo tudi, da ob uporabi sistemskega programskega vmesnika uporabljamo podatkovne tipe sistema. Zaradi tega NativeScript omogoča uporabo najnovejših funkcionalnosti in domorodnih knjižnic, ne da bi bilo za to potrebno pisanje dodatnih knjižnic ali čakanje na novo različico ogrodja NativeScript.

5.2 Arhitektura aplikacije

Arhitektura aplikacije, zgrajene z ogrodjem NativeScript, je odvisna od uporabljenega aplikacijskega ogrodja. Za aplikacije zgrajene brez dodatnega aplikacijskega ogrodja, je priporočena uporaba arhitekturnega vzorca MVVM.

Aplikacije, zgrajene z uporabo aplikacjskega ogrodja Vue.js ali Angular, ki je bil uporabljen za razvoj aplikacije v tej nalogi, pa imajo komponentno zgradbo. Komponente so posamezni deli aplikacije, ki vključujejo definicijo uporabniškega vmesnika in kode, neposredno povezane s tem kosom uporabniškega vmesnika.

5.3 Razvoj aplikacije

Večina razvoja aplikacij z ogrodjem NativeScript je potekala na računalniku z operacijskim sistemom Windows. Računalnik Mac je bil uporabljen samo za preverjanje delovanja in testiranje aplikacije iOS v simulatorju iOS. Za razvoj sta bila uporabljena urejevalnika Visual Studio Code in PhpStorm z nameščenimi razširitvami za NativeScript. Visual Studio Code je Microsoftov brezplačni urejevalnik kode, ki omogoča namestitev mnogih razširitev in se s tem približuje integriranemu razvojnemu okolju. PhpStorm pa je integrirano razvojno okolje podjetja JetBrains, namenjeno predvsem PHP ali spletnim razvijalcem, vendar ga je, tudi zaradi mnogih razširitev, mogoče uporabiti tudi za druge namene.

5.3.1 Postavitev delovnega okolja

NativeScript za delovanje uporablja Node.js. Node.js je izvajalno okolje za programski jezik JavaScript, kar pomeni, da omogoča izvajanje programov JavaScript izven brskalnikov. Za namestitev ga prenesemo s spletne strani nodejs.org. Priporočena je uporaba zadnje dolgo-podprte različice. Ogrodje namestimo iz ukazne vrstice z ukazom *npm install -g nativescript*, na sistemu Mac OS moramo ukaz pognati skupaj z ukazom sudo. Stikalo -g pomeni, da bomo NativeScript namestili globalno. Po namestitvi lahko uporabljamo ukaz tns, ki je kratica za Telerik NativeScript. Potrebna je še namestitev nastavitev za razvoj na izbranih platformah (iOS in Android).

Na računalnike Windows lahko namestimo samo nastavitve za platformo Android. To storimo z ukazom: *@powershell -NoProfile -ExecutionPolicy*

Bypass -Command "iex ((new-object net.webclient).DownloadString('https://-/www.nativescript.org/setup/win'))".

Na računalnikih Mac lahko namestimo nastavitve za obe platformi, za pravilno namestitev platforme iOS pa mora biti na računalniku nameščeno razvojno okolje XCode. Namestitev poženemo z ukazom: *ruby -e "$(curl -fsSL https://www.nativescript.org/setup/mac)"*.

Ko je namestitev zaključena, lahko poženemo ukaz *tns doctor*, ki nam pove, če je vse pripravljeno, kot mora biti oz. nam prikaže navodila za odpravo napak.

Visual Studio Code prenesemo s spletne strani code.visualstudio.com in ga namestimo oz. odpakiramo. Poženemo ga in namestimo razširitev za razvoj aplikacij NativeScript. To storimo z izbiro menijske opcije "View" in izberemo "Extensions". V iskalnik vnesemo "nativescript" in ga namestimo z gumbom "Install".

5.3.2 Kreiranje novega projekta

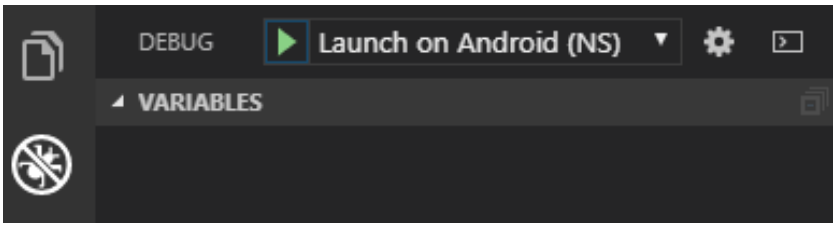
Za večino operacij v NativeScript uporabimo poganjanje tns ukazov v ukazni vrstici. Tako je tudi z ustvarjanjem novega projekta oz. aplikacije.

V ukazni vrstci navigiramo do želene mape na disku. Za ustvarjanje praznega projekta poženemo ukaz *tns create ImeAplikacije*, lahko pa se odločimo za uporabo katere izmed pripravljenih predlog ali predloge, ki jo z opcijo *–template* prenesemo iz repozitorija git. Za našo aplikacijo uporabimo predlogo, ki vsebuje enostavno aplikacijo narejeno z uporabo aplikacijskega ogrodja Angular, zato poženemo ukaz *tns create ns-notes-app –ng*.

Mapo lahko nato odpremo v urejevalniku Visual Studio Code in pričnemo z razvojem. Z desnim klikom na drevesno strukturo projekta in izbiro možnosti "Open in Terminal" lahko odpremo ukazno vrstico zntoraj urejevalnika. Z ukazom *npm i nativescript-angular-cli* lahko namestimo NativeScript Angular CLI, s katerim lahko na enostavnejši način ustvarjamo nove komponente in storitve.

Z ukazom *tns run Android* lahko poženemo aplikacijo. Če uporabimo

ukaz brez dodatnih argumentov, se bo aplikacija pognala na vseh priključnih napravah in prižganih emulatorjih. Za zagon aplikacije v načinu za razhroščevanje pa v levem meniju izberemo gumb z ikono hrošča, izberemo eno izmed konfiguracij (zagon ali priključitev na pognano aplikacijo) in kliknemo gumb z zeleno ikono trikotnika (slika 5.1). Ko je aplikacija pognana, se bo



Slika 5.1: Zagon razhroščevalnika v urejevalniku Visual Studio Code.

ob shranjevanju sprememb v datotekah aplikacija samodejno ponovno zgradila in osvežila. Za razvoj je priporočljiva uporaba Android emulatorja in ne fizične naprave, saj emulator omogoča zamenjavo samo spremenjenih datotek in zato ni potrebna ponovna gradnja celotne aplikacije.

5.3.3 Webpack

Ob prvem zagonu aplikacije smo opazili, da je izgradnja in zagon precej počasen. Aplikacija je skoraj brez vsebine, zagon na emulatorju Android pa je trajal več kot pet minut. Razlog za to je, da tudi prazna NativeScript aplikacija vsebuje veliko število modulov, ki se nahajajo v mapi node_modules in so potrebni za razvoj. Ob zagonu z ukazom *tns run* se vsi ti moduli zapakirajo v paket skupaj z aplikacijo, ki se nato namesti na napravi ali emulatorju.

Rešitev je uporaba Webpacka. Webpack deluje kot povezovalnik. Pred gradnjo aplikacije se sprehodi skozi uvožene reference v projektu in v proces gradnje vključi samo tiste, ki so dejansko uporabljene, tako dobimo bistveno manjšo aplikacijo. Namestimo ga tako, da poženemo ukaz *npm install –save-dev nativescript-dev-webpack*. Namestitev doda tudi nekaj dodatnih odvisno-

sti, ki jih moramo namestiti, zato poženemo *ukaz npm install*.

Vtičnik uporabimo tako, da ob zagonu aplikacije dodamo opcijo *–bundle*, ki jo lahko uporabimo tudi ob gradnji aplikacije: *tns build Android|ios –bundle*.

Še ena uporabna opcija za gradnjo produkcijskega paketa, ki jo pridobimo z uporabo Webpacka, je *–env.uglify*, ki jo lahko uporabimo, da naredimo zgrajen paket človeku neberljiv in ga s tem zaščitimo.

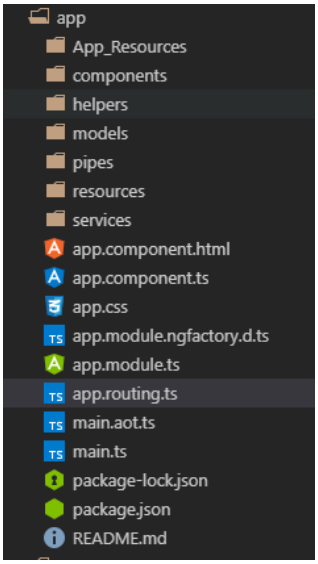
Zaradi uporabe programskega jezika TypeScript, ki se ob gradnji aplikacije prevede v JavaScript, proces gradnje brez uporabe Webpacka poleg vsake datoteke v projektu s končnico .ts doda prevedeno datoteko s končnico .js, kar je precej moteče. Z uporabo Webpacka se tega znebimo, saj poskrbi, da se zgenerirane datoteke shranijo v posebno mapo in ne nazaj v projekt.

5.3.4 Zgradba projekta

Za nas je predvsem pomembna mapa app, kjer se nahaja implementacija naše aplikacije in datoteka package.json, v kateri so definirane odvisnosti. Ostale mape so: hooks (v njej se nahajajo točke, ki jih lahko uporabimo za spremembo procesa gradnje ali poganjanja aplikacije), node_modules (v njej se nahaja izvorna koda nameščenih paketov in vtičnikov) in platforms (v njej se nahajajo datoteke, ki jih zgenerira proces gradnje za posamezno platformo).

V mapi app imamo mapo items, ki jo je tja dodal proces ustvarjanja projekta z uporabo predloge. To mapo odstranimo in v datotekah app.module.ts in app.routing.ts pobrišemo vse uporabe razredov, ki so se v njej nahajali. Zaradi uporabe aplikacijskega ogrodja Angular postavimo datotečno zgradbo po vzorcu Angular (slika 5.2).

- **App_Resources:** V tej mapi se nahajajo viri za posamezne platforme. Tukaj za obe platformi nastavimo ikone in zagonske zaslone.
- **components:** V tej mapi se nahajajo komponente, v katerih je definiran uporabniški vmesnik.



Slika 5.2: Datotečna zgradba aplikacije NativeScript Angular.

- **helpers:** Vsebuje implementacije pomožnih metod, kot so npr. pretvorniki.
- **models:** Vsebuje razrede objektov ali entitet.
- **pipes:** Vsebuje implementacije transformacij ali pretvornikov, namenjenih uporabi v definiciji uporabniških vmesnikov.
- **resources:** Vsebuje vire, kot so ikone in slike. Uporabljeni viri se ob procesu gradnje skopirajo v mapo virov na posamezni platformi.
- **services:** V tej mapi se nahaja implementacija večine logike aplikacije.

Mapa se imenuje services (sl. storitve), ker Angular prihaja z vgrajeno podporo za vbrizgavanje odvisnosti (ang. Dependency Injection). Razred, označen z dekoratorjem @Injectable(), pa se imenuje storitev (ang. Service).

Datoteka **app.module.ts** vsebuje definicijo modula oz. v našem primeru celotne aplikacije. V njej je definirana vstopna točka aplikacije, uvozi modulov in deklaracije komponent in ponudnikov oz. storitev. V datoteki

app.routing.ts so nastavljene poti, ki navigatorju povedo, katero komponento mora pokazati ob navigaciji na določen naslov. Ostali pomembni datoteki sta še **app.css**, ki vsebuje slog aplikacije in **app.component.ts**, ki je vstopna točka aplikacije.

5.3.5 Gradnja uporabniškega vmesnika

Uporabniški vmesnik v NativeScript z uporabo ogrodja Angular gradimo s komponentami. Komponente so sestavljene iz treh datotek:

- datoteke HTML, ki vsebuje definicijo uporabniškega vmesnika, napisano v označevalnem jeziku XML,
- datoteke CSS, ki vsebuje slog komponente,
- datoteke TypeScript, ki vsebuje programski razred komponente. Uporabniški vmesnik, definiran v datoteki HTML, ima dostop do javnih spremenljivk ter metod v tem razredu in jih lahko uporabi za prikaz vrednosti ter izvajanje akcij ob interakciji uporabnika z zaslonom.

Gradniki, ki jih lahko uporabimo za gradnjo uporabniškega vmesnika v predlogi XML, so zelo podobni gradnikom v Xamarin Forms in jih lahko razdelimo v tri kategorije:

- **Page** ali stran. Predstavljajo en zaslon.
- **Layout** ali razvrščevalnik. Uporabljajo se za razvrščanje elementov na zaslon. V NativeScript obstajajo naslednji razvrščevalniki:
  - FlexboxLayout – deluje podobno kot Flexible Box Layout v CSS-ju,
  - AbsoluteLayout – razvršča elemente glede na absolutno določeno pozicijo,
  - DockLayout – razvršča elemente po zunanjih robovih, preostali prostor lahko zasede tretji dodan element,

  - GridLayout – razvršča elemente v mrežo, enako kot v Xamarin Forms to izvaja Grid,
  - StackLayout – sklada elemente horizontalno ali vertikalno,
  - WrapLayout – sklada elemente horizontalno ali vertikalno s prelivanjem, enako kot v Xamarin Forms to počne FlexLayout.

- **UI Widget** ali vizualni gradnik. V to kategorijo spadajo vsi ostali gradniki (gumbi, vnosna polja, drsni pogled, seznam ...).

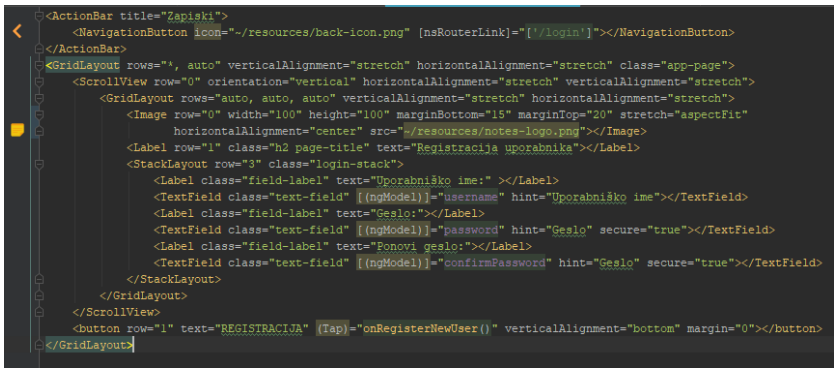
Strani z uporabo ogrodja Angular ne potrebujemo. Za posamezne komponente lahko kot osnovni element uporabimo kar vizualni gradnik ali katerega izmed razvrščevalnikov. Uporaba usmerjevalnika (ang. Router) za navigacijo na določeno komponento pa bo izbrano komponento zapakirala v stran in jo prikazala.

Primer za dodajanje zaslona za registracijo

Za dodajanje komponente lahko uporabim NativeScript Angular CLI in poženemo ukaz: *tns generate component Registration*. NativeScript Angular CLI, za razliko od Angular CLI, zgenerirane komponente ne doda samodejno v polje deklaracij v datoteki app.module.ts. To moramo storiti ročno. Prav tako ob generiranju komponente ne omogoča izbire podmape, v kateri želimo, da se komponenta nahaja, zato moramo zgenerirano mapo registration, ki se nahaja v mapi app, ročno premakniti v mapo components. V datoteki app.routing.ts v polje routes dodamo zapis "{ path: 'registration', component: RegistrationComponent }". Sedaj lahko za navigacijo na zaslon za prijavo uporabimo metodo navigateByUrl na usmerjevalniku z argumentom "registration" ali pa vizualnim gradnikom v kodi XML nastavimo [nsRouterLink]="['/registration']", v tem primeru se bo navigacija izvedla ob kliku na gradnik.

Visual Studio Code, kljub nameščenemu vtičniku, v datotekah HTML/XML ne prepozna vizualnih gradnikov NativeScript in njihovih lastnosti. Zato je

bilo za gradnjo uporabniškega vmesnika uporabljeno razvojno okolje PhpStorm. Na sliki 5.3 je prikazana vsebina datoteke HTML komponente zaslona za registracijo. Poseben gradnik zaslona je ActionBar, ki ga dodamo



Slika 5.3: Koda XML komponente zaslona za registracijo.

izven razvrščevalnika in predstavlja navigacijsko vrstico. Razred komponente se nahaja v datoteki .ts in deluje podobno kot pogledni model. Za zaznavanje sprememb lastnosti skrbi ogrodje Angular, zato za osvežitev zaslona ob spremembah ni potrebna dodatna logika. Na sliki 5.4 je prikazan razred komponente RegistrationComponent, kjer je vidna tudi uporaba vgrajenega vbrizgavanja odvisnosti.

Uporabljene razširitve oz. komponente

Dodatne pakete, razširitve in vtičnike lahko v aplikacijo namestimo z uporabo urejevalnika paketov npm (ang. Node Package Manager). Za namestitev paketa uporabimo ukaz *npm install 'ime-paketa'* oz. skrajšano *npm i 'ime-paketa'*. Za izdelavo grafičnega vmesnika sta bila uporabljena dva paketa:

- **nativescript-ui-sidedrawer**: NativeScript nima vgrajene podpore za navigacijski predal, kot jo ima Xamarin Forms z uporabo strani MasterDetailPage, zato je bil uporabljen vtičnik NativeScript UI SideDrawer. Po namestitvi je treba pred uporabo v modul aplikacije dodati uvoz modula NativeScriptUISideDrawerModule z dodajanjem zapisa v polje

```
@Component({
    moduleId: module.id,
    selector: 'app-registration',
    templateUrl: './registration.component.html',
    styleUrls: ['./registration.component.css']
})
export class RegistrationComponent implements OnInit {
    username: string;
    password: string;
    confirmPassword: string;

    constructor(private _userService: UserService,
                private _dialogService: DialogService,
                private _router: Router) { }

    ngOnInit() { }

    public onRegisterNewUser() {
```

Slika 5.4: Implementacija razreda komponente RegistrationComponent.

imports. Uporabimo ga kot osnovni gradnik ali razvrščevalnik v komponenti zaslona, na katerem ga želimo prikazati. Vsebuje lahko dva elementa. Element z vsebino zaslona označimo z atributom tkMainContent, element z vsebino predalnika pa s tkDrawerContent. Lahko ga uporabimo kot osnovni element komponente AppComponent, vendar bo potem predalnik dosegljiv na vseh zaslonih, tudi na zaslonu za prijavo in registracijo, česar pa ne želimo. V naši aplikaciji je bila vsebina predalnika premaknjena v svojo komponento, predalnik pa je bil uporabljen v komponentah zaslonov za prikaz seznama zapiskov in urejanje zapiska.

- **nativescript-floatingactionbutton**: Paket vsebuje vizualni gradnik imenovan NativeScript-FloatingActionButton, ki je bil uporabljen za prikaz plavajočega gumba za dodajanje zapiska.

Prikaz dialogov in indikatorja obdelave

Dialoge lahko v NativeScript prikazujemo z uporabo razreda Dialogs. Problem pa je, da tem dialogom ne moremo nastaviti barve ozadja, slog gumbov in besedila pa je na njih prepisan z za aplikacijo globalno določenim slogom. Rešitev je, da vsebino v komponenti AppComponent, ki vsebuje samo vtičnico za usmerjevalnik, zavijemo v GridLayout, kamor dodamo še indikator obdelave in vsebino dialoga. V razredu komponente AppComponent nato poslušamo na dogodka showMessageDialog$ in setBusyIndicator$ v za ta namen napisani storitvi, imenovani DialogService. Storitev poleg teh dveh dogodkov vsebuje tudi metodi *setBusy*, ki sprejme argument tipa boolean, in *showMessageDialog*, ki sprejme niz naslova, besedila in gumba dialoga. Storitev je zaradi uporabe vbrizgavanja odvisnosti dosegljiva v celotni aplikaciji.

Slog aplikacije

Podobno kot v Xamarin Forms lahko tudi v NativeScript v definiciji XML uporabniškega vmesnika nastavimo vrednosti posameznih lastnosti (barva ozadja, barva besedila ...). Namesto nastavljanja vsake lastnosti posebej lahko z lastnostmi CSS elementom nastavimo lastnost style. Lahko pa uporabimo datoteke CSS ali SCSS, kjer določimo razrede in jih na elementih uporabimo z nastavitvijo lastnosti class. Postopek je enak kot pri razvoju spletnih aplikacij. V datoteki app.css se nahajajo globalni razredi, ki jih lahko uporabimo v vseh komponentah, poleg tega pa lahko ima tudi vsaka komponenta svojo datoteko CSS, katere razredi pa so dosegljivi samo znotraj te komponente. Za selektorje v datotekah CSS lahko uporabimo imena razredov vizualnih gradnikov in s tem nastavimo slog vsem vizualnim gradnikom določenega tipa znotraj območja vidnosti. V datotekah CSS lahko uporabimo atribute ([]), identifikatorje (#), psevdo selektroje (:) in hierarhične selektorje (' ', >, +). NativeScript trenutno podpira 49 različnih lastnosti CSS. Za nastavljanje specifičnega sloga za določeno platformo lahko uporabimo tri ločene datoteke .css: app-common.css (vsebuje razrede slogov skupnih obema platformama), app.ios.css (vsebuje za platformo iOS splecifične ra-

zrede slogov) in app.android.css (vsebuje za platformo Android specifične razrede slogov). Imena razredov v datotekah common.css, ios.css in android.css se lahko prekrivajo. Drug način za določanje platformno specifičnih slogov pa je uporaba vtičnika nativescript-platform-css, ki nam omogoča uporabo razredov .ios in .android v hierarhičnih selektorjih.

Zagonski zaslon in ikone aplikacije podobno kot v Xamarinu nastavimo za vsako platformo posebej. Razlika je v tem, da v NativeScript nimamo posebnih projektov za iOS in Android, ampak mapo App_Resources, v kateri se nahajajo viri ločeno za Android in iOS. Postopek nastavljanja zagonskih slik in ikon je enak kot v Xamarinu, s to razliko, da v NativeScript nimamo možnosti uporabe grafičnega urejevalnika.

Uporaba ikon in slik: Slike in ikone, ki jih v NativeScript prikazujemo s pomočjo elementa Image, lahko shranimo v katerokoli mapo. V naši aplikaciji je temu namenjena mapa resources. Za prikaz slike elementu Image nastavimo lastnost src s potjo do ikone ali slike, kar prikazuje slika 5.5.

```
<Image row="1" width="100" height="100" marginBottom="30" stretch="aspectFit"
       horizontalAlignment="center" src="~/resources/notes-logo.png"></Image>
```

Slika 5.5: Prikaz slike z uporabo elementa Image.

Komunikacija med zasloni

Preproste argumente lahko z uporabo usmerjevalnika podamo zaslonu, na katerega navigiramo, z uporabo iskalnih parametrov v naslovu. Te parametre v komponenti zaslona preberemo z uporabo storitve ActivatedRoute. Za prenos kompleksnejših objektov lahko implementiramo lastno storitev, v katero nastavimo objekt, zaslon, ki objekt potrebuje, pa ga iz nje prebere.

Za prikaz modalnega okna pa lahko uporabimo storitev ModalDialogService in njeno metodo *showModal*, ki kot argumente sprejme komponento, ki jo želimo prikazati, in dodatne možnosti, kamor lahko nastavimo tudi kompleksne objekte. Dodatne možnosti dobimo kot argument tipa ModalDialogParams v konstruktorju razreda komponente modalnega okna. Ta argument

uporabimo tudi za zapiranje modalnega okna, tako da pokličemo metodo *closeCallback* in ji podamo rezultat.

5.3.6 Delo s podatkovno bazo SQLite

Junija letos se je na GitHubu pojavila različica knjižnice TypeORM s podporo za NativeScript, ki jo je razvil githubov uporabnik championswimmer. Zadnja različica paketa TypeORM te podpore še nima vključene, lahko pa jo namestimo neposredno iz GitHuba z uporabo ukaza:
npm i typeorm@git+https://github.com/championswimmer/typeorm.git#-npm_package_nativescript. Po namestitvi knjižnice pa se aplikacija ni več pognala, zato knjižnica ni bila uporabljena.

Za delo s podatkovno bazo SQLite je bil tako uporabljen vtičnik NativeScript sqlite, nameščen z ukazom *tns plugin add nativescript-sqlite*. Vse metode vtičnika so asinhrone, uporabimo pa jih lahko s pomočjo povratnih klicev (ang. Callback) ali obljub (ang. Promise). Uporaba povratnih klicev lahko privede do globokih gnezdenj in nepregledne kode. V aplikacijah NativeScript je zaradi najboljše podpore na platformah Android in iOS, priporočena uporaba ES5 ciljne različice JavaScript. Ta pa za delo z obljubami nima neposredne podpore uporabe operatorjev *async* in *await*, zato je treba za njuno uporabo spisati pomožne metode. Lahko pa v datoteki tsconfig.json nastavimo vrednost zastavice "noEmitHelpers" na *false*. S tem načinom bo prevajalnik sam zgeneriral pomožne metode za vsako uporabo operatorjev *async* in *await*.

Novo oziroma obstoječo podatkovno bazo inicializiramo s konstruktorjem *new Sqlite*, ki mu kot argument podamo ime datoteke podatkovne baze, kot je prikazano na sliki 5.6. Za izvajanje ukazov SQL uporabimo metodo *execSQL*, za branje podatkov pa metodo *get*, ki ji podamo stavek SQL SELECT in argumente, ki jih v stavku SELECT določimo z znakom '?'. Vse metode za delo s podatkovno bazo se v naši aplikaciji nahajajo v razredu DatabaseService.

```
const db = await new Sqlite('NotesNotifications.db').catch(err => {
    throw (ErrorType.dbOpenDbError);
});
await db.execSQL(`CREATE TABLE IF NOT EXISTS Notification (
    Id INTEGER PRIMARY KEY AUTOINCREMENT,
    User TEXT NOT NULL,
    NoteId INTEGER NOT NULL
    );`).catch(err => {
        throw (ErrorType.dbCreateTablesError);
});

this._notificationDatabase = db;
this._notificationDatabase.resultType(Sqlite.RESULTSASOBJECT);
```

Slika 5.6: Inicializacija podatkovne baze za obvestila.

5.3.7 Uporaba spletne storitve REST

Oblika zapisa podatkov JSON je v programskem jeziku TypeScript, ki je nadnabor programskega jezika JavaScript, neposredno podprta, saj je to, kot že ime pove, oblika zapisa objektov v jeziku JavaScript. V aplikaciji NativeScript Angular lahko za komunikacijo s strežnikom uporabimo razred HttpClient, ki se nahaja v paketu @angular/common/http. Razred omogoča uporabo metod HTTP na generičen način. Za klic metode HTTP POST tako uporabimo metodo *post* v razredu HttpClient z nastavljenim podatkovnim tipom odgovora. Primer uporabe je na sliki 5.7. Vse metode, namenjene

```
public synchronize(notes: Array<WsNote>, lastSyncDateTime: string): Observable<WsResponse<WsSynchronization>> {
    const requestData =  new WsSynchronization();
    requestData.lastSyncDateTime = lastSyncDateTime;
    requestData.notes = notes;
    return this._http.post<WsResponse<WsSynchronization>>(this._syncMethodUrl, requestData, this.getHttpOptions());
}
```

Slika 5.7: Uporaba razreda HttpClient za komunikacijo s spletno storitvijo REST.

komunikaciji s spletno storitvijo REST, so v naši aplikaciji implementirane v razredu NotesWebServiceManagerService. V razredu se nahaja tudi privatna metoda *getHttpOptions*, ki vrača zaglavne podatke zahteve, kamor nastavi ContentType z vrednostjo "application/json" in Authorization z vrednostjo

avtentikacijskega žetona uporabnika. HttpClient za zahteve ne omogoča nastavitve časovne omejitve, zato je bila za preverjanje dosegljivosti strežnika uporabljena metoda *race* v razredu Promise. Metoda kot argument sprejme polje funkcij in se zaključi, ko prva izmed funkcij zaključi izvajanje. Podamo ji funkcijo, ki uprabi metodo *ping* in funkcijo, ki čaka tri sekunde.

5.3.8 Varno shranjevanje podatkov

Za varno shranjevanje podatkov je bil uporabljen vtičnik NativeScript Secure Storage plugin, nameščen z ukazom *tns plugin add nativescript-secure-storage*. Vtičnik za varno shranjevanje na platformi iOS na fizičnih napravah uporablja knjižnico SAMKeychain, ki shranjuje podatke v iOS Keychain, na simulatorju iOS pa podatke shranjuje v NSUserDefaults, kar pomeni, da lahko aplikacijo še naprej razvijamo brez nakupa računa Apple Developer. Za varno shranjevanje na platformi Android vtičnik uporablja knjižnico Hawk, ki za šifriranje in shranjevanje podatkov uporablja knjižnico Facebook conceal. Vtičnik shranjuje podatke na način ključ-vrednost. Za shranjevanje sta na voljo metodi *set* in *setSync*, ki kot argumenta sprejmeta ključ in vrednost. Podatke lahko preberemo z metodama *get* in *getSync*, za brisanje zapisov pa lahko uporabimo metodi *remove* in *removeSync*, ki jima podamo ključ zapisa, ki ga želimo izbrisati, ali metodi *removeAll* in *removeAllSync*, ki izbrišeta vse zapise naše aplikacije.

5.3.9 Dodajanje slik

Za izbiranje slik je bil uporabljen vtičnik NativeScript Image Picker, nameščen z ukazom *tns plugin add nativescript-imagepicker*. Vtičnik je sicer enostaven za uporabo vendar pa bi dokumentacija zanj lahko bila boljša. Brez pregledovanja vsebine nameščenega paketa bi bilo zato zelo težko ugotoviti podatkovni tip (polje objektov ImageAsset) obljube, ki jo vrača metoda *present*, ki jo uporabimo za prikaz izbirnika slik. Uporabo vtičnika prikazuje slika 5.8. Vrnjen rezultat najprej z uporabo razreda ImageSource pretvorimo

```
var context = Imagepicker.create({
    mode: "single"
});
context.authorize()
    .then(function() {
        return context.present();
    }).then(selection => {
        selection.forEach((imageAsset: ImageAsset) => {
            ImageSource.fromAsset(imageAsset).then(res => {
                const base64 = res.toBase64String("jpeg", 100);
                const attachment = new NoteAttachment();
                attachment.content = base64;
                attachment.type = "jpeg";
                if(!this.note.attachments) this.note.attachments = new Array<NoteAttachment>();
                this.note.attachments.push(attachment);
            });
        });
    }).catch(err =>  {
    this._dialogService.showMessageDialog("Napaka", "Prišlo je do nepričakovane napake.");
});
```

Slika 5.8: Uporaba vtičnika NativeScript Image Picker za izbiro slike.

v instanco tega razreda in nato v niz Base64, ki je znakovna predstavitev polja zlogov. Niz shranimo v podatkovno bazo, za prikaz slike pa ga ponovno pretvorimo v slikovni vir (ImageSource), ki ga s podatkovno vezavo nastavimo kot vrednost lastnosti src vizualnega gradnika Image.

5.3.10 Lokalna obvestila

Tudi za prikaz lokalnih obvestil je bil uporabljen vtičnik, in sicer NativeScript Local Notifications Plugin. Nameščen je bil z ukazom *tns plugin add nativescript-local-notifications*. Pred uporabo je treba v NgModule v datoteki app.module.ts dodati polje entryComponents z zapisom SetNotificationComponent. Vtičnik prihaja z odlično dokumentacijo dosegljivo na GitHub repozitoriju [13], kjer se nahaja tudi primer uporabe v aplikaciji NativeScript Angular.

V naši aplikaciji se metode za delo z lokalnimi obvestili nahajajo v razredu NotificationService, kjer se ob zagonu aplikacije registrira tudi metoda, ki se izvede ob povratnem klicu, ko uporabnik klikne na obvestilo. Na sliki 5.9 je prikazana uporaba vtičnika za nastavljanje obvestila.

```
await LocalNotifications.schedule([{
    id: getIdRes.result,
    title: title,
    body: text,
    badge: 1,
    ongoing: false,
    at: date
}]).catch(err => {
    throw ErrorType.notificationSchedulingError
});
```

Slika 5.9: Uporaba vtičnika NativeScript Local Notifications Plugin za nastavljanje obvestila.

5.3.11 Lokalizacija

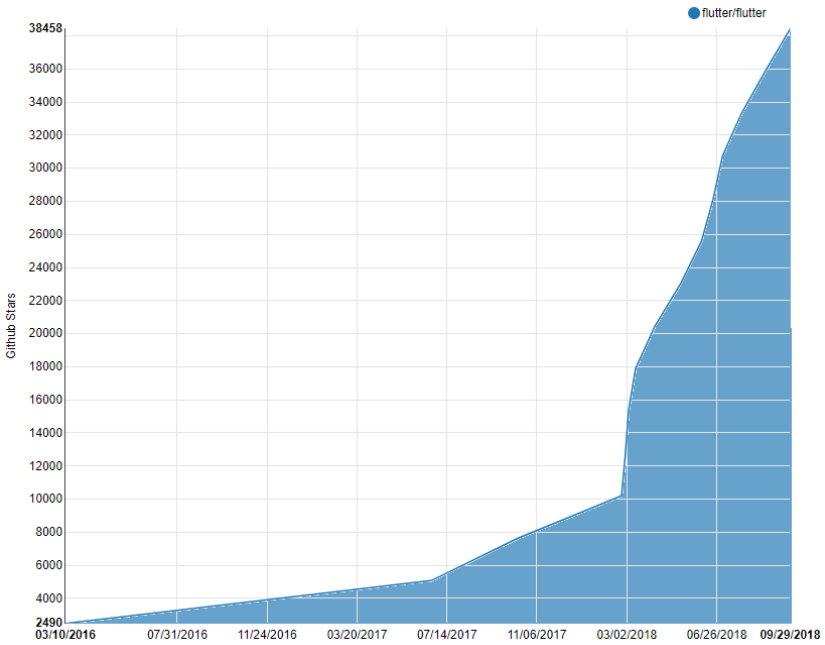
Za lokalizacijo obstaja v repozitoriju urejevalnika paketov npm kar nekaj knjižnic, vendar večina izmed njih za izbiro jezika uporablja jezik, nastavljen v napravi, in ne omogoča dinamičnega spreminjanja jezika medtem ko aplikacija teče. Primer knjižnice, ki to omogoča, je NGX-translate. Po namestitvi paketa je treba v modul aplikacije v polje imports dodati TranslateModule in mu nastaviti nalagalnik, ki pove, v kateri mapi se nahajajo prevodi. Za prevode uporabimo datoteke .json, ki jih poimenujemo z okrajšavami imen jezikov, katerih prevode vsebujejo npr. "sl.json" in "en.json". Do prevodov lahko dostopamo tako, da ob nastavljanju lastnosti elementom v definiciji uporabniškega vmesnika XML uporabimo cev (ang. Pipe): "{{ kljuc | translate }} ali pa v kodi uporabimo storitev TranslateService. Z uporabo storitve lahko nastavimo tudi privzeti in izbrani jezik z metodama *setDefaultLang* in *use*, ki jima kot argument podamo ime datoteke .json.

Poglavje 6

Flutter

Flutter se je na začetku imenoval Sky. Pod tem imenom je bil predstaljen na dogodku Dart Developer Summit aprila leta 2015 kot Googlov eksperimentalni projekt, ki omogoča uporabo programskega jezika Dart za razvoj aplikacij za platformo Android. Na tej predstavitvi je bil poudarek na hitrem razvoju in tekočem izvajanju razvitih aplikacij s konsistentnim izrisovanjem 120 sličic na sekundo. Novembra leta 2015 se je Sky preimenoval v Flutter. Na dogodku Google I/O leta 2017 je bil Flutter prvič predstavljen kot zgodnja alfa različica, 27. februarja letos (2018) pa je izšla prva beta različica. Sledili sta različici beta 2, ki je poenostavila namestitveni proces in beta 3, ki je prinesla podporo za Androidov Material design. Pred kratkim (19. 9. 2018) pa je izšla že druga različica predogleda izdaje, ki naj bi bila tudi zadnja pred izidom različice 1.0.

Flutter omogoča razvoj aplikacij za platformi iOS ter Android in je osnovni način razvoja aplikacij za nov Googlov operacijski sistem Fuchsia, ki naj bi bil namenjen namestitvi na vseh vrstah naprav. Čeprav je Flutter novo ogrodje, pa ga nekatera podjetja že uporabljajo za razvoj produkcijskih mobilnih aplikacij, med njimi je tudi Google. Na sliki 6.1 je prikazan graf števila zvezdic repozitorija GitHub ogrodja Flutter po dnevih. Aplikacije z ogrodjem Flutter razvijamo v programskem jeziku Dart, ki ga je razvil Google z namenom, da bi nadomestil programski jezik JavaScript. Aplikacije napisane v tem jeziku



Slika 6.1: Graf števila zvezdic repozitorija GitHub ogrodja Flutter po dnevih [12].

lahko izvajamo v Dart izvajalnem okolju, lahko jih prevedemo v programski jezik JavaScript (spletne aplikacije), lahko pa jih tudi prevedemo v strojno kodo. Avgusta letos je izšla druga različica jezika Dart, ki je od takrat tudi privzeta različica za razvoj aplikacij Flutter. Glavna novost nove različice je, da prinaša trdnost podatkovnih tipov (ang. Type Soundness).

6.1 Način delovanja

Flutter za izvajanje aplikacije na posamezni platformi uporablja lupino domorodne aplikacije (ang. Shell). Lupina ima, kot vsaka domorodna aplikacija popolni dostop do sistemskega programskega vmesnika. V lupini teče pogon, napisan v programskem jeziku C++, ki vsebuje grafični pogon Skia, izvajalno okolje Dart in kanale metod (ang. Method Channels), ki služijo kot povezava med sistemskim programskim vmesnikom in aplikacijo Flutter.

Za dostop do platformno specifičnih funkcionalnosti, ki v Flutter niso vgrajene, ali za uporabo domorodnih knjižnic lahko implementiramo lastni kanal metode. To storimo tako, da v lupini aplikacije implementiramo metodo s funkcionalnostjo, ki jo želimo, v domorodnem programskem jeziku (Java, Kotlin, Objective-C, Swift) in jo registriramo kot kanal metode z izbranim imenom. Do metode lahko potem v aplikaciji Flutter dostopamo z objektom razreda MethodChannel, ki ga skonstruiramo z imenom registriranega kanala metode.

Za izris uporabniškega vmesnika Flutter ne uporablja domorodnih elementov ampak lastne, ki jih izrisuje z uporabo grafičnega pogona Skia. Princip delovanja je bolj podoben delovanju mobilne igre, kot aplikacije. Slika 6.1 prikazuje diagram komponent aplikacije Flutter.

Slika 6.2: Diagram komponent aplikacije Flutter [10].

6.2 Arhitektura aplikacije

Arhitektura aplikacije Flutter ni specifično določena. Za večje aplikacije pa je zaradi preglednosti, priporočena uporaba arhitekturnega vzorca MVP. MVP: Model-View-Presenter je podobno kot MVVM sestavljen iz treh komponent:

- model – je vmesnik, ki določa podatke, ki jih pogled prikazuje;
- pogled (ang. View) – je pasiven element, ki samo prikazuje podatke modela in preusmerja uporabnikove ukaze na voditelja;
- voditelj (ang. Presenter) – obdeluje uporabnikove ukaze in pripravlja podatke modela.

V aplikaciji Flutter sicer popolna implementacija arhitekturnega vzorca MVP ni mogoča, saj mora pogled še vedno izvajati nekatere operacije, kot je na primer osveževanje stanja.

6.3 Razvoj aplikacije

Podobno kot razvoj aplikacije NativeScript, je tudi razvoj aplikacije Flutter večinoma potekal na računalniku z operacijskim sistemom Windows, računalnik Mac pa je bil uporabljen le za testiranje aplikacije iOS. Flutter za razvoj omogoča uporabo razvojnega okolja Android Studio ali urejevalnika Visual Studio Code. Na računalniku Windows je bilo za razvoj uporabljeno razvojno okolje Android Studio, na računalniku Mac pa Visual Studio Code z nameščenim vtičnikom Flutter.

6.3.1 Postavitev delovnega okolja

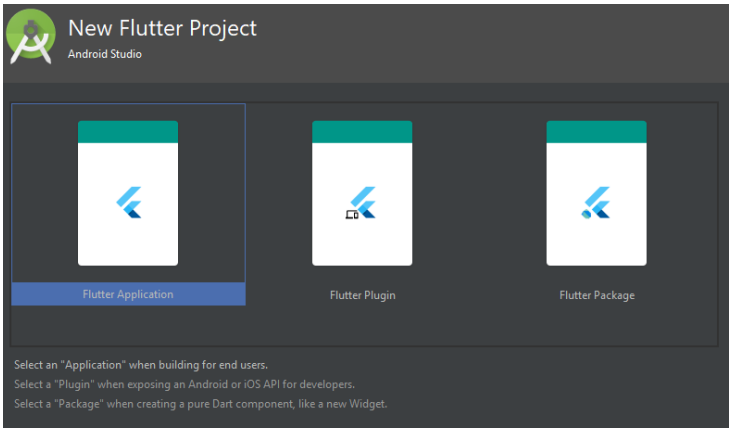
Flutter za delovanje potrebuje git. Če ta na računalniku ni nameščen, ga prenesemo s spletne strani git-scm.com in namestimo. Nato s spletne strani flutter.io prenesemo paket .zip Flutter SDK za operacijski sistem na katerega ga nameščamo. Razpakiramo ga na želeno lokacijo na disku npr. C:\src\Flutter. Če želimo ukaz *flutter* uporabljati v ukazni vrstici, moramo v spremenljivko "Path", dodati pot do lokacije Flutter SDK. Na računalniku Windows to naredimo tako, da odpremo urejevalnik okoljskih spremenljivk ("Control Panel ->User Accounts ->User Accounts ->Change my environment variables") in lokacijo izvršljive datoteke, ki se nahaja v mapi Flut-

ter, dodamo na konec, ločeno s podpičjem. Na računalniku Mac to storimo z ukazom *export PATH=‘pwd‘/flutter/bin:$PATH*. Sedaj lahko v ukazni vrstici poženemo ukaz *flutter doctor*, ki nam pove, kaj nam še manjka, preden lahko pričnemo z razvojem. Tudi Flutter za razvoj aplikacij iOS na računalniku Mac potrebuje nameščeno razvojno okolje XCode. Za razvoj aplikacij za platformo Android pa moramo namestiti Android Studio, tudi če ga ne bomo uporabljali za razvoj. Prenesemo ga s spletne strani developer.android.com/studio/ in ga namestimo s pomočjo namestitvenega čarovnika.

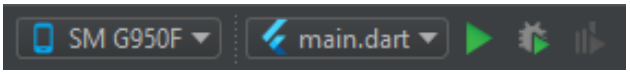
6.3.2 Kreiranje novega projekta

Nov projekt najhitreje ustvarimo z ukazom *flutter create ime_aplikacije*. Ime aplikacije lahko vsebuje samo male črke, številke in podčrtaje, ne sme se začeti s številko ali končati s podčrtajem. Drug način je uporaba urejevalnika Visual Studio Code, kjer iz menija izberemo opcijo "View ->Command Palette", nato v iskalnik vnesemo "flutter" in s seznama izberemo "Flutter: New Project", vnesemo ime aplikacije in pritisnemo tipko "Enter", nazadnje še izberemo lokacijo na disku. Tretji način pa je uporaba razvojnega okolja Android Studio, kjer iz menija izberemo opcijo "File ->New ->New Flutter Project". Odpre se okno, kjer izberemo vrsto projekta. Možni so: "Flutter Application", "Flutter Plugin" (namenjen razvoju vtičnika za dostop do sistemskega programskega vmesnika platforme) in "Flutter Package" (namenjen razvoju Dart komponent oz. paketov) (slika 6.3). V naslednjih korakih vnesemo ime aplikacije, izberemo lokacijo na disku in ime paketa aplikacije Android, vključimo lahko tudi podporo programskih jezikov Swift in Kotlin za razvoj platformno specifičnih funkcionalnosti.

Ustvarjen projekt vsebuje enostavno aplikacijo, ki jo lahko poženemo z izbiro naprave, zagonske datoteke in klikom gumba z ikono zelenega trikotnika ali zelenega trikotnika s hroščem za zagon v načinu za razhroščevanje, ki se nahaja v orodni vrstici zgoraj desno (slika 6.4). Pognana aplikacija se bo samodejno osvežila ob shranjevaju spremenjenih datotek.



Slika 6.3: Izbira vrste projekta ob kreiranju novega projekta z uporabo razvojnega okolja Android Studio.



Slika 6.4: Zagon aplikacije v razvojnem okolju Android Studio.

6.3.3 Zgradba projekta

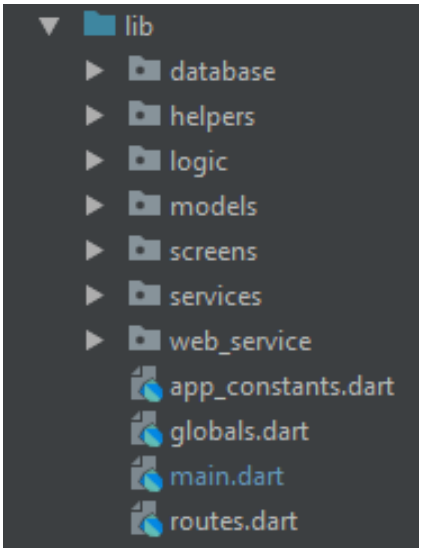
Ustvarjen projekt vsebuje štiri mape:

- android – vsebuje vstopno točko aplikacije Android in njene nastavitve,
- ios – vsebuje vstopno točko aplikacije iOS in njene nastavitve,
- lib – vsebuje celotno skupno implementacijo aplikacije,
- test – namenjena implementaciji testov.

Za nas je pomembna še datoteka pubspec.yaml, v kateri so definirane odvisnosti oz. uporabljeni paketi.

Končna zgradba mape lib je prikazana na sliki 6.5:

- mapa database – vsebuje implementacijo razreda za delo s podatkovno bazo SQLite,



Slika 6.5: Datotečna zgradba mape lib.

- mapa helpers – vsebuje implementacije pomožnih metod,
- mapa logic – vsebuje implementacijo logike za upravljanje z uporabniki, zapiski in obvestili,
- mapa models – vsebuje razrede modelov oz. entitet,
- mapa screens – vsebuje implementacijo uporabniškega vmesnika oz. zaslonov,
- mapa services – vsebuje implementacije storitev,
- mapa web_service – vsebuje implementacijo logike za komunikacijo s spletno storitvijo REST,
- datoteka app_constants.dart – vsebuje konstante aplikacije,
- datoteka globals.dart – vsebuje globalne spremenljivke in dogodke,
- datoteka main.dart – vsebuje metodo *main*, ki je vstopna točka aplikacije Flutter,

- datoteka routes.dart – vsebuje poti, ki jih lahko uporabimo za navigacijo z uporabo naslova zaslona

6.3.4 Gradnja uporabniškega vmesnika

Za razliko od Xamarin Forms in NativeScript, Flutter nima vgrajene podpore uporabe označevalnega jezika za gradnjo uporabniških vmesnikov. Uporabniški vmesnik aplikacije Flutter gradimo s sestavljanjem elementov v programskem jeziku Dart. Lahko pa si pomagamo z grafičnim urejevalnikom na spletni strani flutterstudio.app, ki deluje po sistemu "vleci in spusti" in v ozadaju generira kodo zaslona Dart. V ogrodju Flutter vsi gradniki uporabniškega vmesnika izhajajo iz osnovnega vizualnega gradnika (ang. Widget). Tudi celotna aplikacija je vizualni gradnik. Obstajata dve vrsti vizualnih gradnikov:

- **Stateless widget**: vizualni gradnik brez stanja,
- **Stateful widget**: vizualni gradnik s spreminjajočim se stanjem.

Vizualni gradniki brez stanja so tisti, ki se jim skozi življenski cikel stanje ne spreminja. Tak primer so statični napisi in gumbi. Vizualni gradniki s spreminjajočim se stanjem pa so tisti, ki se jim stanje skozi življenski cikel lahko spremeni. Tak primer je vnosno polje ali zaslon aplikacije z drsnim premikanjem. Da sprožimo ponovni izris gradnika s spreminjajočim se stanjem, moramo lastnosti stanja spreminjati s pomočjo metode *setState*. Gradnike, vgrajene v ogrodje Flutter bi lahko razvrstili v pet skupin:

- **osnovni gradniki**:
  - WidgetsApp – osnovni gradnik aplikacije,
  - MaterialApp – osnovni gradnik aplikacije, prirejen za Material design,
  - AppBar – navigacijska vrstica Material design,

  - BottomNavigationBar – navigacijska vrstica Material design na spodnjem delu zaslona,
  - TabBar – vrstica z zavihki Material design,
  - Drawer – navigacijski predal Material design,
  - Scaffold – osnova za zaslon Material design (kot lastnosti omogoča nastavitev elementov Material design),
  - CupertinoNavigationBar – navigacijska vrstica Cupertino design,
  - CupertinoTabBar – Cupertino design vrstica z zavihki,
  - CupertinoTabScaffold – osnova za zaslon Cupertino design (kot lastnosti omogoča nastavitev elementov Cupertino design).

- **razvrščevalniki** (uporabimo jih lahko tako v Material design elementih, kot tudi v Cupertino desing elementih):
  - Row – razvršča elemente v vrsto,
  - Column – razvršča elemente v stolpec,
  - Stack – sklada elemente na kup v smeri Z-osi,
  - IndexedStack – sklada elemente na kup in prikazuje z indeksom določenega otroka,
  - GridView – razvršča elemente v mrežo in omogoča drsno premikanje,
  - ListView – razvršča elemente horizontalno ali vertikalno in omogoča drsno premikanje,
  - CustomScrollView – razvršča elemente glede na uporabniško določen način in omogoča drsno premikanje (otroci morajo biti elementi z osnovnim razredom Sliver).

- **pozicijski elementi** (v ogrodju Flutter so tudi pozicijski elementi vizualni gradniki):
  - Padding – določa odmik vsebine,

  - Center – postavi vsebino na sredino,
  - Align – določa pozicijo (levo, desno, spodaj, zgoraj).

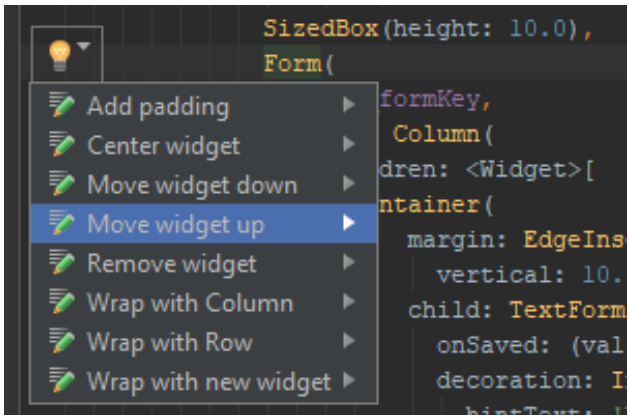
- **vizualni elementi**: vsi ostali gradniki, ki jih prikazujemo na zaslonu (vnosna polja, gumbi ...),
- **ostalo**: tudi elementi, kot so animacije in slog, se v ogrodju Flutter štejejo med vizualne gradnike.

Uporabniški vmesnik z ogrodjem Flutter zgrajene aplikacije se (zaenkrat) ne prilagaja platformi, na kateri aplikacija teče. Odločimo se lahko za izgled aplikacije po oblikovalskih načelih Android (Material design) ali oblikovalskih načelih iOS (Cupertino design). Sicer je možno zgraditi prilagodljiv uporabniški vmesnik, vendar ne brez dodatnega dela. Za našo aplikacijo je bil uporabljen Material design.

Primer za dodajanje zaslona za registracijo

V mapo screens dodamo mapo registration (v razvojnem okolju Android Studio to storimo z desnim klikom na mapo, da odpremo kontekstni meni in nato izberemo opcijo "New ->Package"). V mapo registration dodamo datoteki registration_screen.dart in registration_screen_presenter.dart. V datoteko registration_screen_presenter.dart dodamo dva razreda: RegistrationScreenPresenter, ki vsebuje metodo *registerNewUser*, in abstraktni razred RegistrationScreenContract, ki definira metodi *onRegistrationSuccess* in *onRegistrationError*. Konstruktor razreda RegistrationScreenPresenter mora sprejeti instanco razreda RegistrationScreenContract. V datoteko registration_screen.dart pa dodamo razred RegistrationScreen, ki razširja razred StatefulWidget, in razred RegistrationScreenState, ki razširja generični razred State z nastavljenim podatkovnim tipom RegistrationScreen, in implementira razred RegistrationScreenContract. V konstruktorju razreda RegistrationScreenState ustvarimo instanco razreda RegistrationScreenPresenter, ki mu kot argument konstruktorja podamo samega sebe (*this*). V razredu RegistrationScreenState prepišemo metodo *build*, v kateri vrnemo s kodo zgrajen

uporabniški vmesnik. Koda uporabniškega vmesnika lahko postane nekoliko nepregledna. Če se odločimo, da želimo en gradnik zaviti v drugega, se včasih težko odločimo, kam moramo postaviti zaklepaje, zato pa si lahko pomagamo z razvojnim okoljem Android studio, kot je prikazano na sliki 6.6. Prepišemo

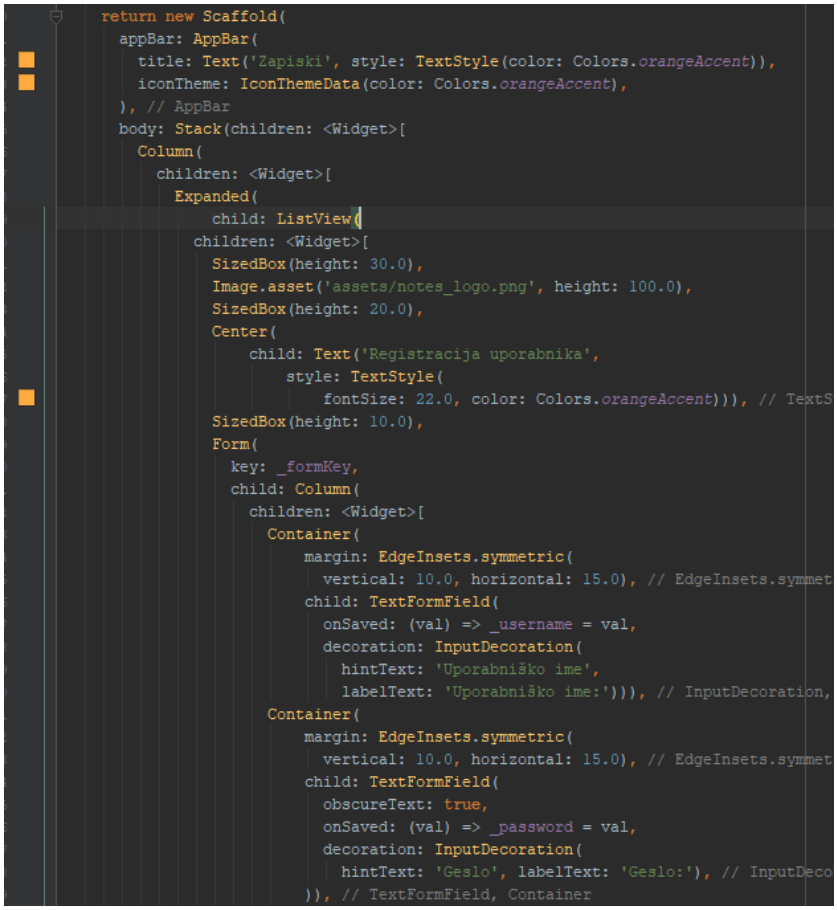


Slika 6.6: Urejanje vizualnega gradnika v razvojnem oklju Android Studio.

še metodi *onRegistrationSuccess* in *onRegistrationError* ter dodamo metodo *_submit*, v kateri preberemo vrednosti vnosnih polj in pokličemo metodo *registerNewUser* v instanci razreda RegistrationScreenPresenter. Na koncu še v datoteki routes.dart v polje routes dodamo zapis, s katerim omogočimo uporabo vgrajenega navigatorja z naslovom. Na sliki 6.7 je prikazan del kode uporabniškega vmesnika zaslona za registracijo.

Prikaz dialogov in indikatorja obdelave

Indikator obdelave je v aplikaciji Flutter implementiran za vsako stran posebej. Uporabljen je vizualni gradnik brez stanja, dodan v razvrščevalnik Stack in prikazan glede na stanje zaslona _isBusy. V aplikaciji Flutter tudi ni bilo problema s slogom vgrajenih dialogov. Da pa se koda za prikaz dialoga ne ponavlja na vsakem zaslonu, je bila implementirana storitev DialogService, ki vsebuje metodo *showAlert*. Flutter oziroma Dart nima vgrajene podpore vbrizgavanja odvisnosti, lahko pa na enostaven način zagotovimo, da bo skozi



Slika 6.7: Del kode uporabniškega vmesnika zaslona za registracijo.

celotno aplikacijo uporabljena samo ena instanca razreda. To storimo tako, da v razred dodamo statično privatno spremenljivko podatkovnega tipa razreda in jo inicializiramo s privatnim konstruktorjem, za javni konstruktor pa uporabimo tovarno (ang. Factory), ki vrne instanco, shranjeno v privatni statični spremenljivki. Takšni razredi so imenovani Singleton razredi.

6.3.5 Slog aplikacije

Kot je bilo že napisano, so v ogrodju Flutter tudi elementi, ki določajo slog, vizualni gradniki. Da nastavimo slog besedila (element Text), mu nasta-

vimo lastnost style z objektom podatkovnega tipa TextStyle. Ponavljajoče se razrede slogov lahko tudi ločimo v svoje razrede.

MaterialApp: Zaradi uporabe razreda MaterialApp za osnovo aplikacije je zelo poenostavljeno nastavljanje barvne sheme in splošne oblike aplikacije. V razredu MaterialApp nastavimo lastnost theme z objektom razreda ThemeData, ki mu nastavimo primarno barvo, barvo poudarka in svetlost. S tem je večina oblikovanja aplikacije že dosežena, poleg tega pa MaterialApp prihaja z velikim številom ikon, ki jih lahko uporabimo za gumbe in ostalo.

Zagonske slike in ikone: Enako kot v NativeScript in Xamarin Forms jih nastavimo za vsako platformo posebej v mapah android in ios.

Uporaba slik: Slike lahko dodamo v mape za vsako platformo posebej ali pa dodamo mapo, npr. assets, katere pot nastavimo v razdelek assets v datoteki pubspec.yaml. Prikažemo jih z gradnikom Image, za katerega uporabimo konstruktor *asset*, ki mu kot argument podamo pot do slike.

6.3.6 Navigacija in komunikacija med zasloni

Za navigacijo lahko uporabimo vgrajen razred Navigator. Na zaslone registrirane v polju routes v datoteki routes.dart lahko navigiramo z naslovom z uporabo metod *pushNamed* ali *pushReplacementNamed*, ki prepreči navigacijo nazaj na prejšnji zaslon. Na zaslone, ki jim želimo podati argument, lahko navigiramo z metodo *push*, ki ji podamo objekt vizualnega gradnika zaslona, ki mu v konstruktorju podamo željene argumente. Flutter ne razlikuje med celozaslonskimi in modalnimi zasloni. Podatke lahko vedno vrnemo z uporabo metode *pop*, ki ji podamo rezultat zaslona. V naši aplikaciji je bil za komunikacijo uporabljen tudi globalni dogodek doRefresh v razredu Globals, na katerega posluša zaslon s seznamom zapiskov.

6.3.7 Delo s podatkovno bazo SQLite

Za delo s podatkovno bazo SQLite uporabimo paket sqflite. Pakete za aplikacije Flutter lahko poiščemo na paketnem repozitoriju, ki se nahaja na spletni

strani pub.dartlang.org/flutter/packages. V aplikacijo dodamo paket tako, da v datoteko pubspec.yaml med odvisnosti oziroma v razdelek dependencies dodamo vrstico z imenom paketa in različico. Za dodajanje sqflite paketa dodamo vrstico: sqflite: ^0.12.1. Shranjevanje datoteke samodejno požene ukaz *flutter packages get*, ki posodobi pakete glede na odvisnosti, določene v tej datoteki.

Uporaba paketa je podobna uporabi vtičnika v aplikaciji NativeScript. Metode za delo s podatkovno bazo so v naši aplikaciji implementirane v razredu DataAccess. Za inicializacijo nove ali obstoječe podatkovne baze najprej z uporabo metode *getDatabasesPath* pridobimo pot do mape, kamor lahko shranjujemo podatke na napravi. Pridobljeni poti dodamo ime datoteke podatkovne baze in uporabimo metodo *openDatabase* za odpiranje podatkovne baze (slika 6.8).

```
Future<Null> initializeNotificationsDatabase() async {
  var databasesPath = await getDatabasesPath();
  String path = join(databasesPath, 'notesNotifications.db');
  Database database = await openDatabase(path, version: 1,
      onCreate: (Database db, int version) async {
    await db.transaction((transaction) async {
      await transaction.execute('''CREATE TABLE IF NOT EXISTS Notification (
              Id INTEGER PRIMARY KEY AUTOINCREMENT,
              User TEXT NOT NULL,
              NoteId INTEGER NOT NULL
            );''');
    });
  });
  _notificationsDatabase = database;
}
```

Slika 6.8: Odpiranje podatkovne baze SQLite z uporabo paketa sqflite.

Vse metode vtičnika so asinhrone. Za izvajanje ukazov SQL so na voljo metode *execute*, *rawInsert*, *rawUpdate* in *rawQuery*. Paket pa vsebuje tudi pomožne metode, ki nam nekoliko olajšajo delo. Da jih lahko uporabimo pa moramo v razredih, ki predstavljajo tabele podatkovne baze, implementirati metodo, ki podatke objekta pretvori v objekt Map, in konstruktor, ki iz objekta Map sestavi objekt razreda. Primer za razred NotesNotification je prikazan na sliki 6.9. Pomožne metode sqflite paketa so:

```dart
class NotesNotification {
  String user;
  int noteId;

  NotesNotification();

  Map<String, dynamic> toDbMap() {
    var map = <String, dynamic>{'User': user, 'NoteId': noteId};
    return map;
  }

  NotesNotification.fromDbMap(Map<String, dynamic> map) {
    user = map['User'];
    noteId = map['NoteId'];
  }
}
```

Slika 6.9: Razred z implementirano metodo *toDbMap* in konstruktorjem *fromDbMap*.

-*insert* (kot argumenta sprejme ime tabele ter objekt vrstice Map in vrne ID vstavljene vrstice),
-*query* (kot argumente sprejme ime tabele, polje imen stolpcev, stavek WHERE ter polje argumentov stavka WHERE in vrne seznam objektov Map),
-*delete* (kot argumente sprejme ime tabele, stavek WHERE ter argumente stavka WHERE in vrne število spremenjenih vrstic),
-*update* (kot argumente sprejme ime tabele, objekt vrstice Map, stavek WHERE ter argumente stavka WHERE in vrne število spremenjenih vrstic).

6.3.8 Uporaba spletne storitve REST

Zahteve HTTP lahko v aplikaciji Flutter izvajamo s pomočjo razreda HttpClient. Razred omogoča pripravo objekta zahteve (HttpClientRequest) z uporabo metod *postUrl* in *getUrl*, ki jima podamo naslov metode na strežniku. Pripravljenemu objektu zahteve lahko nato nastavimo zaglavne podatke in vsebino (ang. Body), zahtevo pa sprožimo z metodo *close*, ki vrne rezultat.

Nekaj dodatnega dela pa imamo s pripravo in branjem podatkov v obliki JSON. Serializacija podatkov v obliko JSON (ali katero drugo obliko zapisa

podatkov v niz znakov, npr. XML) po navadi poteka z uporabo odsevnika, s katerim serializator pridobi imena objektov in spremenljivk. Odsevnik je v programskem jeziku Dart implementiran v knjižnici mirrors. Ta pa je v ogrodju Flutter onemogočena zaradi uporabe statičnega povezovalnika, ki ob gradnji aplikacije zelo učinkovito odstrani vse neuporabljene metode in razrede. To pa z uporabo knjižnice mirrors ne bi bilo mogoče. Za obdelavo podatkov v obliki JSON sta v aplikaciji Flutter na voljo dva načina:

- **ročna serializacija**: Ročna serializacija je primerna za manjše, ne preveč kompleksne objekte. Za ročno serializacijo uporabimo razred JsonCodec. Objekt serializiramo tako, da ga pretvorimo v objekt Map in ga kot argument podamo metodi *encode*. Za deserializacijo uporabimo metodo *decode*, ki vrne objekt Map, ki ga nato pretvorimo v objekt razreda. Podobno kot pri uporabi paketa sqflite, lahko v razrede, ki jih želimo serializirati, dodamo metodo *toJsonMap* in konstruktor *fromJsonMap*. Ročna serializacija je bila v naši aplikaciji uporabljena za serializacijo objekta User, ki se uporablja v metodah *login* in *registerNewUser*.

- **avtomatična serializacija z uporabo generatorjev kode**: Če imamo v aplikaciji večje število kompleksnejših razredov, katerih objekte želimo serializirati, lahko pisanje metod in konstruktorjev za mapiranje postane naporno. Zato lahko uporabimo generatorje kode. V naši aplikaciji je bil za generiranje kode uporabljen paket json_serializable, za zagon generatorja pa potrebujemo še paket build_runner. Oba dodamo med razvojne odvisnosti v datoteki pubspec.yaml (razdelek dev_dependencies), saj ne bosta del končne aplikacije, ampak bosta uporabljena samo za razvoj. V razdelek odvisnosti pa dodamo še paket json_annotation. Razrede, katerih objekte želimo serializirati, označimo z anotacijo (ang. Annotation) @JsonSerializable(), njihove lastnosti pa lahko označimo z anotacijo @JsonKey, s katero določimo ime objekta oz. spremenljivke v zapisu objekta JSON. Razredi morajo imeti konstruktor brez argumentov, nad njih pa moramo dodati še vrstico part, ki

pove, da se del razreda nahaja v drugi datoteki. Primer pripravljenega razreda je prikazan na sliki 6.10. Za generiranje delnih razredov

```dart
part 'ws_sync_response.g.dart';

@JsonSerializable()
class WsSyncResponse {
  int errorType;  //WsErrorType
  String responseMessage;
  String errorMessage;
  WsSynchronization result;

  WsSyncResponse();

  factory WsSyncResponse.fromJson(Map<String, dynamic> json) => _$WsSyncResponseFromJson(json);

  toJson() => _$WsSyncResponseToJson(this);
}
```

Slika 6.10: Razred, ki uporablja zgenerirano kodo za serializacijo v obliko JSON.

poženemo ukaz *flutter packages pub run build_runner build*. V razredih lahko nato uporabimo privatne metode zgeneriranega dela razreda za tovarniški konstruktor in metodo, ki objekt pretvori v niz JSON. Če razrede spremenimo, moramo ponovno zagnati generator, lahko pa ukazu dodamo stikalo *watch* in s tem sprožimo opazovalnik, ki bo samodejno pognal generator, ko bo zaznal spremembe v razredih.

6.3.9 Varno shranjevanje podatkov

Za varno shranjevanje uporabnikovega avtentikacijskega žetona in uporabniškega imena je bil uporabljen paket flutter_secure_storage. Paket na platformi iOS za varno shranjevanje podatkov uporablja iOS KeyChain. Na platformi Android podatke zakriptira z uporabo enkripcije AES, skrivni ključ enkripcije AES nato zakriptira z enkripcijo RSA, katere ključ shrani v Android KeyStore. KeyStore je v Androidu prisoten od različice 4.3 naprej, zato je treba v datoteki android/app/build.gradle nastaviti minSdkVersion na 18 (Android 4.3 je razvit na različici SDK 18).

Za uporabo paketa najprej inicializiramo instanco razreda FlutterSecure-

Storage. Vse metode razreda so asinhrone. Instanco razreda nato uporabimo za shranjevanje podatkov z metodo *write*, ki kot argumenta sprejme niz ključa in niz vrednosti. Za branje podaktov lahko uporabimo metodi *read*, ki kot argument sprejme ključ, ali *readAll*. Varno shranjene podatke izbrišemo z metodama *delete* ali *deleteAll*.

6.3.10 Dodajanje slik

Odličen paket za izbiro in zajem slik v aplikaciji Flutter je Image Picker plugin for Flutter. V odvisnosti v datoteki pubspec.yaml ga dodamo z dodajanjem naslednjega zapisa: image_picker: ^0.4.10. Pred uporabo moramo v datoteko ios/Runner/Info.plist dodati ključe NSPhotoLibraryUsageDescription, NSCameraUsageDescription in NSMicrophoneUsageDescription, katerih vrednosti so opisi, zakaj aplikacija potrebuje dostop do teh funkcionalnosti. Paket je zelo enostaven za uporabo. Metoda *ImagePicker.pickImage*, ki ji kot argument podamo vir slike (galerija ali fotoaparat), prikaže izbirnik slik in vrne objekt podatkovnega tipa File. V naši aplikaciji objekt tipa File pretvorimo v polje zlogov in shranimo v podatkovno bazo. Vizualni gradnik Image ima vgrajeno podporo nalaganja slike iz polja zlogov z uporabo konstruktorja *memory* (slika 6.11).

6.3.11 Lokalna obvestila

Za prikazovanje obvestil je bil uporabljen paket Flutter Local Notifications Plugin. Vsa potrebna logika za uporabo paketa je implementirana v razredu NotificationsLogic. Razred vsebuje tri asinhrone metode:

- *initialize*, ki se pokliče ob zagonu aplikacije in požene inicializacijo podatkovne baze za obvestila ter nastavi nastavitve obvestil in ustvari instanco razreda FlutterLocalNotificationsPlugin,
- *scheduleNotification*, ki se pokliče ob shranjevanju zapiska in nastavi obvestilo na izbran čas,

```
void _addImage() async {
  var image = await ImagePicker.pickImage(
      source: ImageSource.gallery, maxHeight: 1000.0, maxWidth: 1000.0);
  if (image == null) return;

  var attachment = new NoteAttachment();
  attachment.content = image.readAsBytesSync();
  attachment.name = image.path;
  attachment.type = image.path
      .split('.')
      .last;

  setState(() {
    _attachments.add(attachment);
  });
}
Image.memory(_attachments[index].content,
    height: 60.0, fit: BoxFit.scaleDown), // Image.memory
```

Slika 6.11: Izbira slike z uporabo paketa image_picker in prikaz slike iz polja zlogov.

- *_onSelectNotification*, ki se izvede, ko uporabnik klikne na obvestilo, kot argument sprejme ID obvestila. Obvestilo pridobi iz podatkovne baze in če je obvestilo namenjeno trenutno prijavljenemu uporabniku sproži navigacijo na zaslon za urejanje zapiska, ki ga obvestilo zadeva.

6.3.12 Lokalizacija

Flutter ima v razredih WidgetsApp in MaterialApp pripravljeno lastnost localizationsDelegates namenjeno lokalizaciji. Za uporabo te lastnosti moramo med odvisnosti v datoteki pubspec.yaml dodati zapis: flutter_localizations: sdk: flutter in implementirati razreda AppLocalizations in AppLocalizationsDelegate, ki razširja generični razred LocalizationsDelegate z nastavljenim podatkovnim tipom AppLocalizations razreda. Razred AppLocalizations vsebuje metodo *load*, ki naloži izbran jezik in metode *get*, ki vračajo prevedena sporočila. Za generiranje datotek, v katerih so shranjeni prevodi, lahko uporabimo paket intl_translation, ki ga dodamo med razvojne odvisnosti. V razred AppLocalizations dodamo metode *get* za vsa besedila v aplikaciji, nato poženemo ukaz *flutter pub pub run intl_translation:extract_to_arb –output-*

dir=lib/l10n lib/localizations.dart. Zgenerira se datoteka intl_messages.arb, ki jo skopiramo in prevedemo za vsak podprt jezik, v njihovih imenih pa zamenjamo "messages" z imenom jezika, ki mu je datoteka namenjena (npr. intl_en.arb). Da povežemo nalagalnik jezika s prevedenimi datotekami, poženemo še ukaz *flutter pub pub run intl_translation:generate_from_arb –output-dir=lib/l10n \–no-use-deferred-loading lib/localization.dart lib/l10n/intl_*.arb*. Na koncu v razredu aplikacije nastavimo lastnosti supportedLocales s poljem podprtih jezikov in localizationsDelegates, kamor nastavimo polje, ki vsebuje naš razred AppLocalizationsDelegate. Do prevodov dostopamo z metodo *AppLocalizationsDelegate.of(context).kljuc*. Če želimo jezik dinamično spreminjati, moramo razred aplikacije spremeniti v vizualni gradnik s stanjem in spremembo jezika v razredu AppLocalizationsDelegate sprožiti z uporabo metode *setState*.

Poglavje 7

Ugotovitve in primerjava

Na koncu je bila izvedena primerjava uporabljenih ogrodij in z njimi razvitih produktov.

7.1 Primerjava razvoja

Najprej nas je zanimalo, kakšna so posamezna razvojna ogrodja s stališča razvjalca in kako se primerjajo med sabo na različnih področjih.

7.1.1 Xamarin

Razvoj aplikacije z uporabo Xamarina zagotovo ni najpreprostejši. Za sam začetek razvoja je potrebnih kar nekaj korakov (nekaj jih sicer tudi ni nujnih, kot je na primer namestitev knjižnice MvvmCross). Razvoj je omejen na razvojno okolje Visual Studio, kar pa niti ni slabo, saj je Visual Studio zelo dobro razvojno okolje in ponuja odlična orodja za razvoj mobilnih aplikacij. Eno izmed bolj uporabnih orodij je prikaz zaslona simulatorja iOS na razvojnem računalniku Windows. Prednost Xamarina je tudi programski jezik C#.

Med razvojem se kar hitro zgodi, da je za dosego cilja potrebna implementacija platformno specifične funkcionalnosti ali razširitve obstoječih razredov ogrodja. Preko urejevalnika paketov NuGet je na voljo veliko število paketov

in knjižnic, na žalost pa ni veliko dobrih paketov gradnikov uporabniških vmesnikov.

Izmed ogrodij, ki smo jih primerjali, je bil Xamarin najboljši v uporabi podatkovne baze SQLite, v repozitoriju paketov NuGet sta bili na voljo tudi dobri knjižnici za varno shranjevanje podatkov in izbiro slik iz naprave. Tudi lokalizacija aplikacije na opisan način (za katero niti ne potrebujemo zunanje knjižnice) je, po mojem mnenju, najenostavnejša. Med razvojem aplikacije s Xamarin Forms nismo naleteli na nerešljive probleme, zaradi katerih bi se morala aplikacija razlikovati od načrtovane. Za gradnjo uporabniškega vmesnika mogoče včasih zmanjka kakšen gradnik, je pa sam postopek zelo dobro definiran, čeprav, zaradi razvoja po arhitekturnem vzorcu MVVM, tudi kar zahteven.

7.1.2 NativeScript

NativeScript je od Xamarina preprostejši za uporabo. Po kreiranju projekta lahko hitro pričnemo z dodajanjem zaslonov brez dodatnih korakov. Razvoj ni omejen na določeno razvojno okolje, nobeno od uporabljenih (Visual Studio Code in PhpStorm) pa ni ponujalo grafičnega uporabniškega vmesnika za dodajanje komponent in ostalih operacij.

Izgradnja aplikacije Android brez uporabe Webpacka je zelo počasna. Razhroščevalnik v urejevalniku Visual Studo Code pa z uporabo Webpacka ne deluje. Tudi z uporabo Webpacka je živo osveževanje pognane aplikacije precej počasno. Sam razvoj aplikacije z ogrodjem Angular pa je zelo podoben razvoju spletne aplikacije z ogrodjem Angular. TypeScript je sicer boljši jezik kot JavaScript, vendar določanje podatkovnih tipov še vedno ni obvezno, kar pa je sicer lahko težava predvsem ob uporabi zunanjih knjižnic. V repozitoriju urejevalnika paketov npm je na voljo veliko število vtičnikov in knjižnic, ne delujejo pa vsi v aplikaciji NativeScript, pri paketih vizualnih gradnikov pa je treba biti pazljiv, da paket podpira tudi aplikacijsko ogrodje, ki ga uporabljamo (v našem primeru Angular).

NativeScript je omogočal najenostavnejšo uporabo spletne storitve REST,

v repozitoriju paketov npm pa so bile na voljo tudi dobre knjižnice za varno shranjevanje podatkov, izbiro slik iz naprave in prikaz lokalnih obvestil. Med razvojem aplikacije NativeScript smo imeli ves čas probleme, ki so nas upočasnjevali pri razvoju in so se po navadi na koncu kar sami od sebe razrešili (npr. prihaja do napake ob gradnji aplikacije, nekaj poskusov kasneje pa napake ni več). Pisanje platformno specifične kode pa je zelo nepriročno, saj moramo na nek način uporabljati TypeScript/JavaScript za pisanje kode v drugem jeziku.

7.1.3 Flutter

Flutter je zelo dobro podprt v razvojnem okolju Android Studio. Od vseh treh ogrodij, ki smo jih primerjali, je najenostavnejši za uporabo.

Živo osveževanje pognane aplikacije je zelo hitro tako na emulatorjih in napravah Android, kot tudi na simulatorju iOS, hiter in dober je tudi razhroščevalnik v razvojnem okolju Android Studio. Sestavljanje uporabniškega vmesnika v kodi lahko privede do nepreglednosti. Pri gradnji uporabniškega vmesnika pa smo imeli več težav zaradi nepoznavanja gradnikov, ki se od gradnikov v Xamarin Forms in NativeScript kar precej razlikujejo, kot pa s preglednostjo kode. Podpora Material design izgledu aplikacije je zelo dobra in bi jo težko ločili od izgleda domorodne aplikacije Android. Zelo hitro lahko sestavimo osnovno obliko zaslona, saj gradnike, kot sta navigacijska vrstica in navigacijski predal, nastavimo kar na osnovni gradnik zaslona. Tudi dokumentacija, dosegljiva na spletni strani flutter.io, je zelo dobra. Za serializacijo podatkov v obliko JSON in pripravo datotek za lokalizacijo je potrebnega nekaj dodatnega dela, vendar sam postopek ni preveč zapleten. V repozitoriju pub paketov se nahaja kar lepo število dobrih paketov kljub temu, da je ogrodje še precej novo. Paketi v repozitoriju so tudi dobro dokumentirani, v iskalniku pa se lahko omejimo samo na pakete Dart, ki delujejo v ogrodju Flutter. Dobra lastnost ogrodja Flutter je tudi programski jezik Dart, ki sicer (še) ni zelo razširjen, omogoča pa hiter in dobro definiran razvoj.

Razvoj aplikacije v ogrodju Flutter je bil najhitrejši kljub nepoznavanju

programskega jezika Dart in vizualnih gradnikov, med razvojem pa ni bilo večjih težav. Še ena dobra lastnost ogrodja Flutter pa je, da platformno specifično kodo pišemo v domorodnem jeziku platforme. Izvajanje aplikacije Flutter v lupini domorodne aplikacije pa omogoča tudi razvoj dela domorodne aplikacije v ogrodju Flutter in s tem prehod domorodne aplikacije v aplikacijo, razvito z ogrodjem Flutter, po korakih.

7.1.4 Primerjava ogrodij po področjih

V naslednji tabeli so podane ocene uporabljenih ogrodij po posameznih področjih. Za posamezno področje je ogrodje lahko dobilo od 1 do 3 točke, kjer predstavlja 3 najboljšo oceno, ogrodja pa so lahko tudi izenačena.

področje/ogrodje	**Xamarin**	**NativeScript**	**Flutter**
postavitev delovnega okolja	3	2	2
razvojna orodja	3	1	2
dokumentacija	2	2	3
programski jezik	3	2	2
razširitve (repozitorij paketov)	2	2	3
gradnja uporabniškega vmesnika	3	2	2
delo s podatkovno bazo	3	1	2
uporaba spletnih storitev	2	3	1
enostavnost razvoja	1	2	3
količina deljene kode	1	2	3
razvoj platformno specifičnih funkcionalnosti	2	1	3
cena	2	3	3
skupaj	27	23	29

Postavitev delovnega okolja nobenega od ogrodij ni težavna, je pa najenostavneša namestitev Xamarina z namestitvenim čarovnikom razvojnega okolja Visual Studio. Kot je že bilo omenjeno, Xamarin ponuja najboljša razvojna orodja, sledi pa mu Flutter, ki omogoča razvoj v razvojnem okolju Android Studio. Dokumentacija za vsa ogrodja je dosegljiva na uradnih spletnih straneh, nekoliko boljša pa je dokumentacija za ogrodje Flutter. Programski jezik Dart je boljši od programskega jezika TypeScript, je pa manj razširjen in ga pozna manj razvijalcev, zato sta ogrodji NativeScript in Flutter v tem področju izenačeni. Za vsa ogrodja je na voljo repozitorij za razširitve (Xamarin: NuGet, NativeScript: npm, Flutter: pub). Repozitorij paketov pub, ki vsebuje pakete za programski jezik Dart, pa omogoča najboljše filtriranje paketov za ogrodje Flutter, na njem pa se, kljub temu, da je ogrodje Flutter še precej novo, nahaja veliko število dobrih paketov z dobro dokumentacijo in primeri uporabe. Ogrodje Flutter sicer ne omogoča uporabe označevalnega jezika za gradnjo uporabniškega vmesnika, ima pa zato veliko število predpripravljenih osnovnih gradnikov, kar omogoča hitro in enostavno oblikovanje aplikacije. Xamarin in NativeScript omogočata uporabo označevalnega jezika, prednosti Xamarina pa sta dobra podpora jezika XAML in dobra razvojna orodja. Za Xamarin je na voljo najboljša knjižnica za delo s podatkovno bazo SQLite, na drugem mestu je knjižnica sqflite za ogrodje Flutter zaradi možnosti uporabe pomožnih metod. NativeScript omogoča uporabo spletnih storitev REST s prenosom podatkov v obliki JSON brez dodatnih knjižnic, največ dela za pripravo podatkov v obliki JSON imamo v ogrodju Flutter. Razvoj aplikacije s Xamarinom je najkompleksnejši, najpreprostejši pa je razvoj z ogrodjem Flutter. Ob uporabi Xamarina najhitreje pridemo do situacije, ko je potreben razvoj platformno specifičnih funkcionalnosti, posledica tega pa je, da se količina kode, deljene med platformami, zmanjšuje. Ogrodje Flutter za uporabniški vmesnik ne uporablja domorodnih vizualnih gradnikov, izgled aplikacije je praktičo enak na obeh podprtih platformah, zato tudi ni potrebe po razvoju platformno specifičnih slogov. Za veliko platformno specifičnih funkcionalnosti lahko na

paketnih repozitorijih najdemo razširitve za vsa tri ogordja. Najboljši način za razvoj teh funkcionalnosti pa je na voljo za ogrodje Flutter, saj omogoča uporabo domorodnih programskih jezikov. Ogrodji NativeScript in Flutter sta brezplačni, Xamarin pa je del razvojnega okolja Visul Studio, ki pa je brezplačno samo za nekomercialne namene.

7.2 Primerjava aplikacij

Zelo pomembno pa je tudi, kako se obnašajo aplikacije, razvite z različnimi ogrodji. Razvite aplikacije so si med seboj kar podobne, kar je bil tudi namen. Testiranje aplikacij je večinoma potekalo na fizični napravi Android (Samsung Galaxy S8), za testiranje aplikacije iOS pa smo uporabili simulator iOS.

Najslabša je aplikacija razvita z ogrodjem NativeScript. Ko je nekaj časa odprta, postane počasna, gumbi v navigacijskem predalu pa postanejo neodzivni. Če kot priponko dodamo zapisku sliko, uporaba aplikacije postane skoraj nemogoča. Poleg tega pa na iPhone X, ki ima posebno obliko zaslona zaradi zareze za kamero in zvočnik, doda bel rob, kar je v nasprotju z Applovimi smernicami (zareza naj se upošteva kot del zaslona) razvoja aplikacij za iPhone X. Včasih pa se nepravilno izrisuje tudi navigacijska vrstica. Aplikaciji razviti z ogrodji Flutter in Xamarin sta po hitrosti delovanja približno enaki, vendar je aplikacija razvita z ogrodjem Flutter po občutku nekoliko bolj tekoča in konsistentna. V aplikaciji Flutter so lepše tudi animacije gumbov in prehodi med zasloni, katerih animacije so v aplikaciji Xamarin izključene, saj na platformi Android niso bile povsem tekoče.

Velika razlika je tudi v velikosti paketa aplikacije, kar pa je večinoma odvisno od velikosti izvajalnega okolja, ki je zapakirano poleg aplikacije:

- Xamarin Forms: 17MB,
- NativeScript: 12MB,
- Flutter: 7MB.

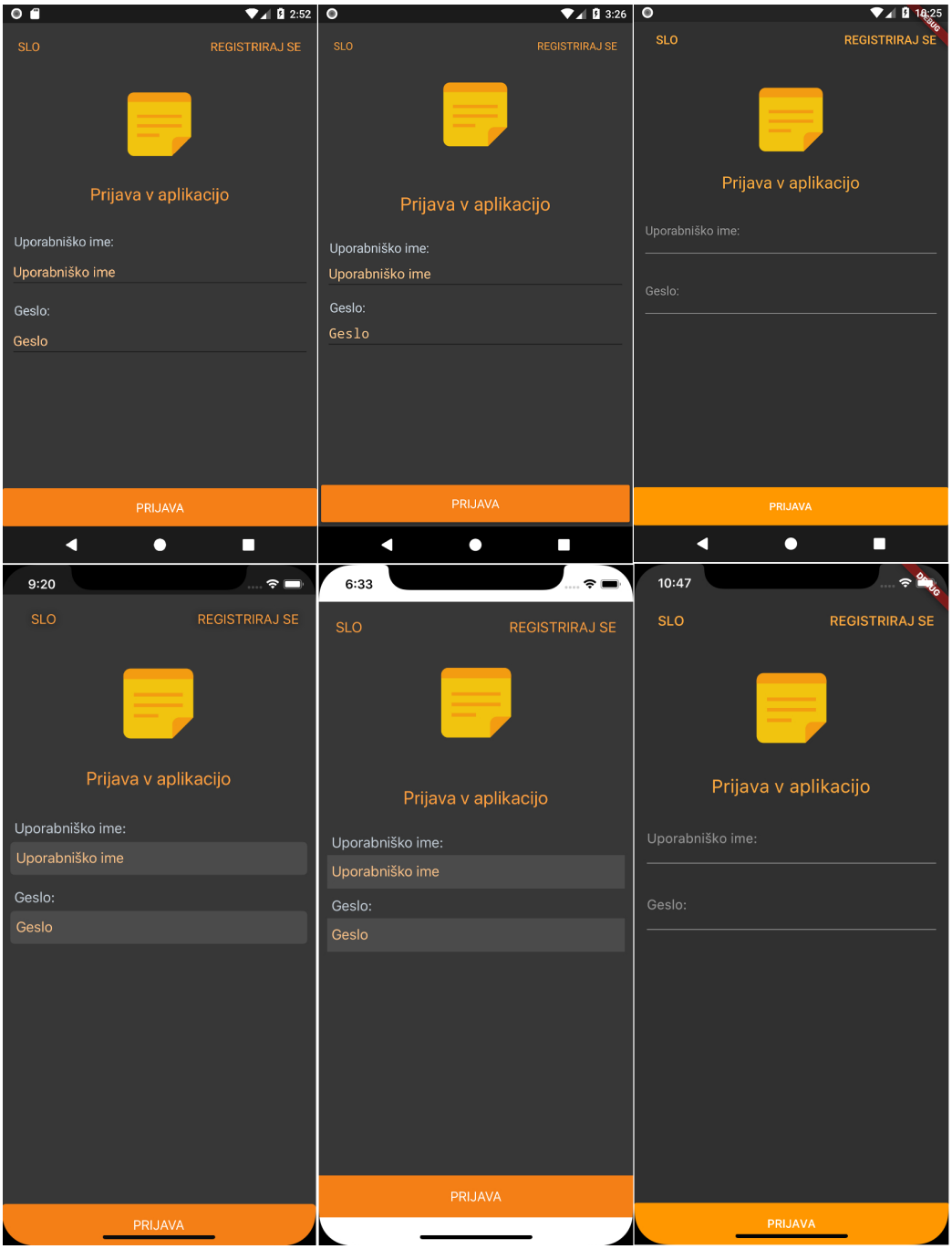
Vizualni izgled aplikacij lahko vidimo na slikah 7.1, 7.2, 7.3 in 7.4. Na slikah lahko vidimo različne zaslone aplikacije, razvite s tremi različnimi ogrodji. Vidimo, da so si zasloni zgrajeni s tremi ogrodji zelo podobni, le zasloni aplikacije, zgrajene z NativScript, imajo na platformi iOS bel okvir, opazen pa je tudi rob okrog gumba na platformi Android, ki ga ni bilo mogoče odstraniti.

7.2.1 Uporabniški test

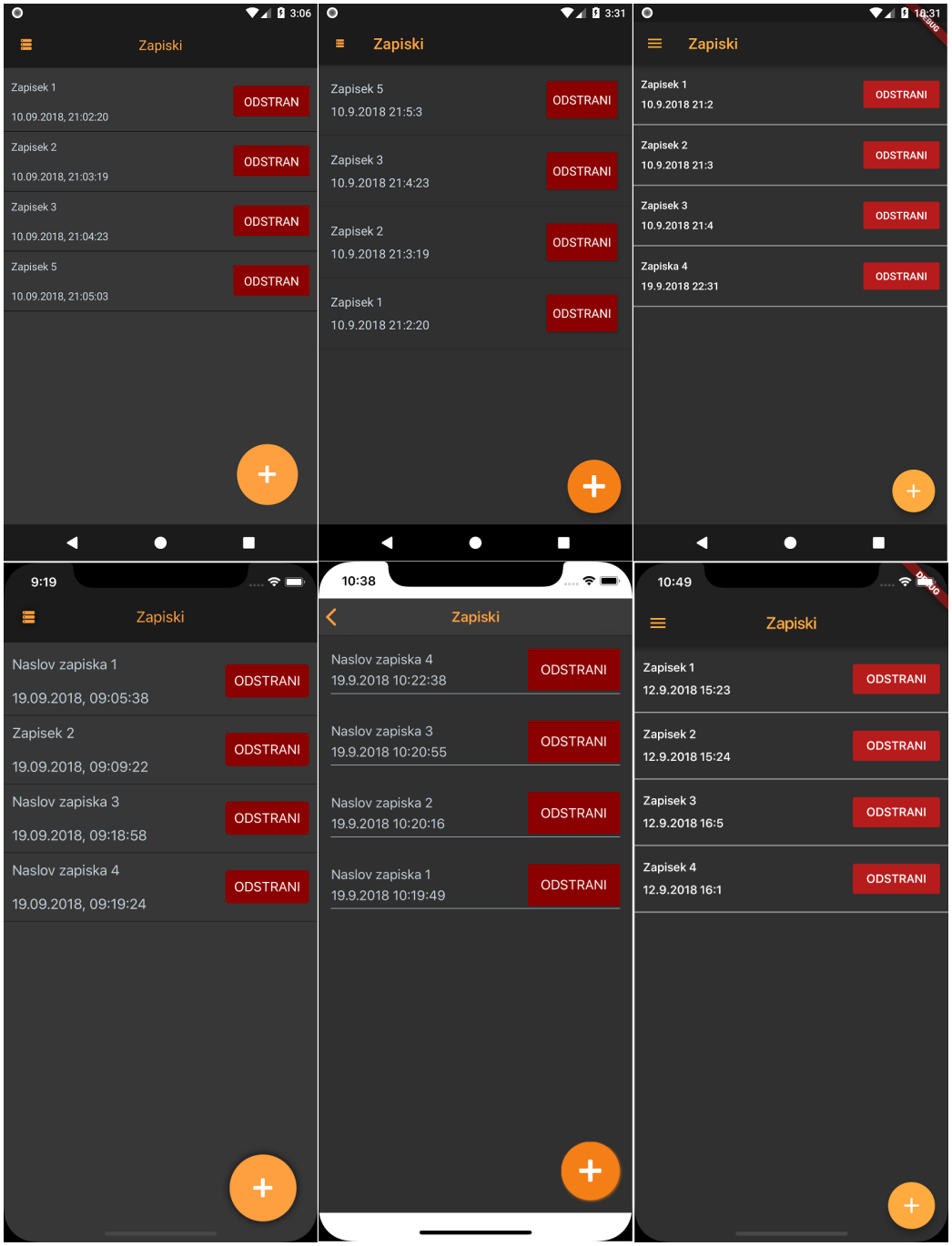
Za uporabniški test pa sem prosil tudi sodelavce, prijatelje in družino. Uporabniški test je bil izveden samo na aplikacijah Android, ker za testiranje aplikacij iOS potrebujemo račun Apple Developer. Naloga uporabnika je bila, da se registrirajo in preiskusijo vse funkcionalnosti (dodajanje zapiska, dodajanje priponke na zapisek, nastavitev obvestila, brisanje zapiska in sinhronizacija zapiskov med aplikacijami) v vseh treh aplikacijah. Odziv ni bil ravno najboljši, rezultat pa vseeno relativno konsistenten. Od sedmih uporabnikov, ki so se odzvali na prošnjo za test, so štirje podali enak rezultat: najbolj jim je bila všeč Flutter aplikacija, najmanj pa NativeScript aplikacija. Dva sta se odločila, da je najboljša Xamarin Forms aplikacija in najslabša NativeScript aplikacija. Eden pa je kot najboljšo določil NativeScript aplikacijo, kot najslabšo pa Xamarin Forms aplikacijo. Rezultati so razvidni v naslednji tabeli:

ogrodje/ocena	**3**	**2**	**1**	**skupaj**
Xamarin	2x	4x	1x	14
NativeScript	1x	0x	6x	9
Flutter	4x	3x	0x	18

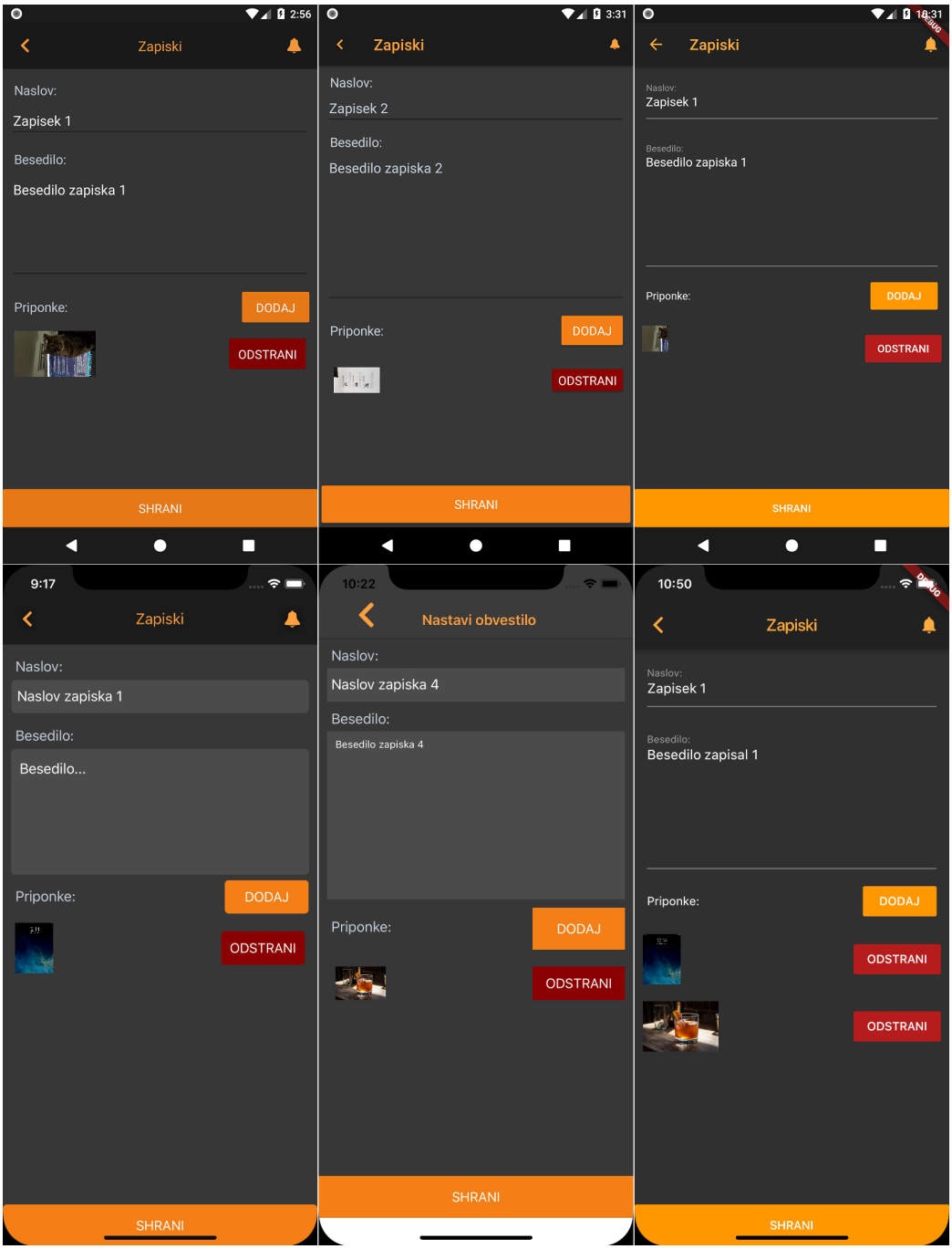
Nobeden izmed uporabnikov ni sporočil, da bi katera izmed testiranih aplikacij nepravilno delovala, navkljub težavam, ki smo jih sami opazili z aplikacijo NativeScript. To kaže, da so navkljub razlikam in morda nekoliko slabšemu (počasnejšemu) delovanju razvitih aplikacij, vsa preizkušena ogrodja primerna za hkraten navzkrižni razvoj aplikacij za več platform.



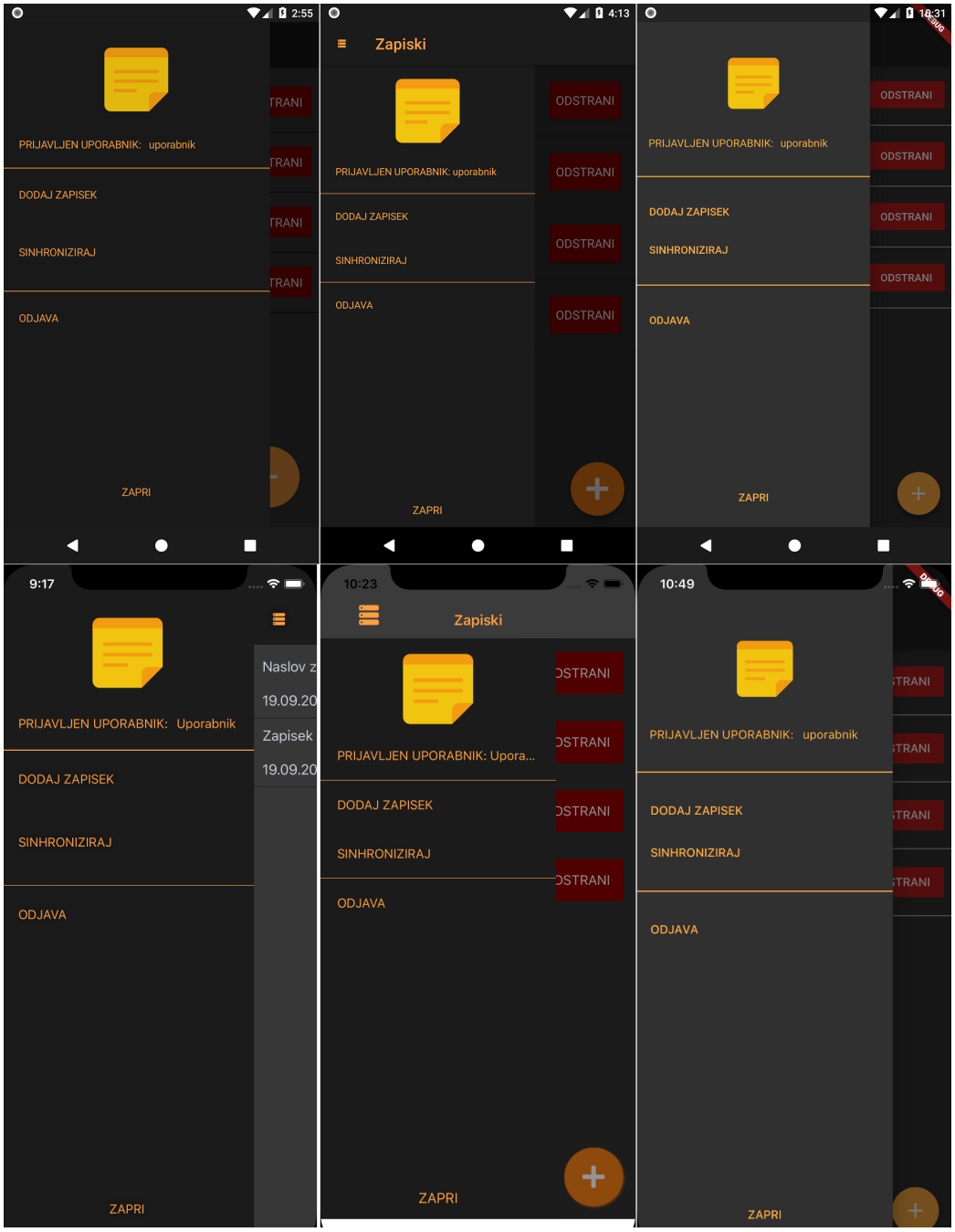
Slika 7.1: Zaslon za prijavo (zgoraj: Android, spodaj: iOS, levo: Xamarin Forms, sredina: NativeScript, desno: Flutter).



Slika 7.2: Zaslon ki prikazuje seznam zapiskov (zgoraj: Android, spodaj: iOS, levo: Xamarin Forms, sredina: NativeScript, desno: Flutter).



Slika 7.3: Zaslon za urejanje zapiska (zgoraj: Android, spodaj: iOS, levo: Xamarin Forms, sredina: NativeScript, desno: Flutter).



Slika 7.4: Navigacijski predal (zgoraj: Android, spodaj: iOS, levo: Xamarin Forms, sredina: NativeScript, desno: Flutter).

Poglavje 8

Sklep

Cilj diplomske naloge je bil raziskati možne pristope k razvoju mobilnih aplikacij, razviti mobilno aplikacijo s tremi ogrodji za hkraten navzkrižni razvoj mobilnih aplikacij za več platform ter primerjati potek razvoja in končnih aplikacij. Cilj je bil v celoti dosežen. V prvem delu smo opisali možne pristope k razvoju mobilnih aplikacij in orodja, ki jih lahko uporabimo za razvoj s posameznim pristopom. Nato so bila izbrana tri ogrodja za hkraten navzkrižni razvoj mobilnih aplikacij za več platform. Ta ogrodja so Xamarin, NativeScript in Flutter. Zastavljena je bila enostavna aplikacija in v celoti razvita z vsemi tremi izbranimi ogrodji. Večji del diplome zajema opis izbranih ogrodij in postopkov razvoja aplikacije z njimi. Na koncu je podana še primerjava razvoja aplikacije, kjer so opisane tudi prednosti in slabosti posameznih ogrodij, ter rezultat uporabniškega testa aplikacij.

Najboljšo uporabniško izkušnjo smo dosegli z aplikacijo, razvito v ogrodju Flutter, ki pa je bilo tudi iz razvijalskega stališča najenostavnejše za uporabo in nam je pri razvoju povzročalo najmanj težav. Aplikacija NativeScript je bila za razvoj najbolj težavna, predvsem zaradi ne najboljših razvijalskih orodij. Aplikacija NativeScript pa se je tudi najslabše odrezala na uporabniškem testu. Xamarin je najkompleksnejši za uporabo, ponuja pa odlična razvijalska orodja in je najprimernejši za razvoj večjih aplikacij. Uporabniški test je postavil aplikacijo, razvito z ogrodjem Xamarin, na drugo mesto.

Literatura

[1] Ionic vs. PhoneGap vs. Framework7. Dosegljivo: https://stackshare.io/stackups/framework7-vs-ionic-vs-phonegap. [Dostopano 24. 9. 2018].

[2] Mobile Operating System Market Share United Kingdom. Dosegljivo: http://gs.statcounter.com/os-market-share/mobile/united-kingdom#monthly-201305-201808. [Dostopano 24. 9. 2018].

[3] Mobile Operating System Market Share Worldwide. Dosegljivo: http://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-201305-201808. [Dostopano 24. 9. 2018].

[4] Number of available applications in the Google Play Store from December 2009 to June 2018. Dosegljivo: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store. [Dostopano 24. 9. 2018].

[5] Number of available apps in the Apple App Store from 2008 to 2018 (in 1,000s). Dosegljivo: https://www.statista.com/statistics/268251/number-of-apps-in-the-itunes-app-store-since-2008. [Dostopano 24. 9. 2018].

[6] Sebastian Cabrera. Why use Xamarin for cross-platform development. Dosegljivo: https://www.uruit.com/blog/2016/09/23/why-choosing-xamarin. [Dostopano 24. 9. 2018].

[7] Brad Umbraugh Craig Dunn. MVVM architecture, ViewModel and LiveData (Part 1). Dosegljivo: https://docs.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/code-sharing. [Dostopano 24. 9. 2018].

[8] Eric Enge. Mobile vs Desktop Usage in 2018: Mobile takes the lead. Dosegljivo: https://www.stonetemple.com/mobile-vs-desktop-usage-study. [Dostopano 24. 9. 2018].

[9] jamesmontemagno. Media Plugin for Xamarin and Windows. Dosegljivo: https://github.com/jamesmontemagno/MediaPlugin. [Dostopano 24. 9. 2018].

[10] Adam Pedley. How Flutter Works. Dosegljivo: https://buildflutter.com/how-flutter-works/. [Dostopano 29. 9. 2018].

[11] Hazem Saleh. Sharing code overview. Dosegljivo: https://proandroiddev.com/mvvm-architecture-viewmodel-and-livedata-part-1-604f50cda1. [Dostopano 24. 9. 2018].

[12] timqian. Flutter GithHub star history. Dosegljivo: http://www.timqian.com/star-history/#flutter/flutter. [Dostopano 29. 9. 2018].

[13] Eddy Verbruggen. NativeScript Local Notifications Plugin. Dosegljivo: https://github.com/EddyVerbruggen/nativescript-local-notifications. [Dostopano 27. 9. 2018].